Polyvariant Strictness Analysis in UHC
M.Sc. Thesis

Augusto Passalaqua Martins
(ICA-3728420)

Department of Computing Science
Utrecht University, The Netherlands

Supervisors:
dr. Jurriaan Hage
dr. Atze Dijkstra

August 31, 2013

Abstract

Strictness analysis detects when it is safe to evaluate expressions before they actually need to be evaluated in a lazy language. We present the development of a polyvariant strictness analysis for Haskell. Previous developments typically used an ad hoc lambda-calculus based language that do not always reflect Haskell's complexity entirely, often lacking support for higher order functions, user defined datatypes, and recursion. Others that did have more extensive language support were mostly monovariant. Our system aims to cover both aspects: a polyvariant system with extensive support for higher order functions and user provided strictness annotations. This is then implemented in the Utrecht Haskell Compiler (UHC) to validate the system and observe its real world effects.

Contents

List of Figures

Chapter 1

Introduction

Lazy evaluation is an evaluation strategy with several interesting qualities. Among other things, it allows the construction of infinite data structures and a straightforward, simple implementation will skip unnecessary computations during evaluation automatically. Those characteristics make it desirable for the programmer, however it also has plenty of drawbacks in terms of performance, particularly when dealing with many delayed but necessary computations. Representations for such delayed computations, called thunks, need to be created and destroyed in a manner that uses more memory and wastes more time with managing those thunks than should be necessary. Detecting when it is safe to evaluate non-lazily a term that represents a computation allows us to avoid those shortcomings.

A single argument function is said to be strict if, whenever it is given a diverging argument, it also diverges. This property is called "strictness." A strict function can be seen operationally as one that evaluates its arguments before returning its result, that is, opposite to how a lazy language works. It should be noted that every strict program that terminates should also terminate if implemented lazily, but not every program that can be constructed in a lazy setting can be fully evaluated strictly correctly. Strictness can also avoid some of the performance penalties introduced by laziness. Therefore, it is beneficial to be able to have a mixed of strict and lazy evaluation on practical, real-world implementations of lazy languages. This mix can be either implemented by means of an explicit encoding of strictness or by automatic analysis. The former is available in Haskell with the *seq* function and the bang pattern. These partially negate the advantage of the lazy environment by giving the programmer such responsibility. Therefore, ideally we would like to have the task of annotating the programs to be performed automatically by the compiler.

This project builds on top of previous work by Holdermans and Hage, and Verburg to implement strictness analysis in the Utrecht Haskell Compiler (**UHC**) [1], [2]. We take these authors' results and expand to include more language constructs, remove the lambda-lifted code restriction, improve higher-order function support with polyvariance, and try to discover the aspects of this analysis that are most important for performance improvements. In the following chapter we present the motivation for solving the problem of strictness through program analysis. Chapter 3 presents a review of some of the already developed strictness analysis techniques and how they achieve some of the aforementioned goals. Afterwards, in Chapter 4, we describe UHC in detail and how its architecture affect implementation choices, while Chapter 5 describes our planned approach. We then describe our type system used for the analysis in Chapter 6, our algorithm for the analysis in Chapter 7 and how that information can be used to lead a UHC code transformation in Chapter 8. Finally, we present the implementation results in Chapter 9 and finish with concluding remarks in Chapter 10.

Chapter 2

Motivation

Lazy evaluation only reduces a given reduceable term as much as necessary. The necessity is driven by the need to inspect a value, such as in a case scrutinee. This means we do not necessarily have terms that are values at once and we might need to store partially unevaluated expressions, called *thunk*s, during program execution. The fact that terms are only evaluated as much as needed allows us to define recursive structures such as infinite lists with no memory penalty other than the description of the computation steps to calculate its as yet non-evaluated part.

For instance, with laziness it is possible to define a list of all naturals in the following fashion:

$$\begin{aligned} naturals = \; & \textbf{let } naturals' \; n = n : (naturals' \; (n+1)) \\ & \textbf{in } \; naturals' \; 0 \end{aligned}$$

Despite being a list with all natural numbers, this list uses finite memory space. Only if we print the value of $naturals$ it will use infinite memory. With lazyness, we may print some of the elements of the list, which will be calculated and with the list expanded until such element.

If we want to pick the 5th element of that list, and assuming previously calculated values to be stored for later sharing, we can imagine the memory representation of the list to render something like this:

$$\begin{aligned} naturals = \; & \textbf{let } naturals' \; n = n : (naturals' \; (n+1)) \\ & \textbf{in } \; 0 : 1 : 2 : 3 : 4 : naturals' \; (4+1) \end{aligned}$$

The in memory representation of the list was expanded until the 5th element, which is the number 4, but the rest of the list was kept in its original representation with the updated n

parameter. Not even the term $(4 + 1)$ before the recursive call needs to be evaluated, since its value is not yet needed.

Suppose now that we have a function such as f below:

$$f = \lambda a\ b\ s \rightarrow \textbf{if}\ s > 0\ \textbf{then}\ a + b\ \textbf{else}\ a$$

In a purely lazy setting values are passed in a "call-by-need" fashion. That means that a, b, and s will not have been necessarily reduced as much as possible before and during f's evaluation. It is possible that these parameters are other functions or expressions and only once the value of f is needed will these be calculated.

For instance, f may be called from another function such as:

$$f' = \lambda c \rightarrow f\ (1 + 2 + 3 + 4)\ 5\ c$$

Regardless of c's value we already know that the expression $(1 + 2 + 3 + 4)$ has to be evaluated. Instead of storing f or f' as a thunk that includes this expression in its unevaluated form we can already perform this computation and store its result. To permit such constructions in Haskell we have a function that is treated differently from regular ones in terms of laziness, the seq function. By use of this function we can rewrite the original f' as:

$$\begin{aligned} f'' = \ & \textbf{let}\ a = (1 + 2 + 3 + 4) \\ & \textbf{in}\ \ seq\ a\ (\lambda c \rightarrow f\ a\ 5\ c) \end{aligned}$$

This form will force the expression $(1 + 2 + 3 + 4)$ to be evaluated before building the inner lambda of the **let** body. Naturally, if f'' itself is never evaluated, this change will have no effect other than increasing our code size and this change becomes detrimental. If it is evaluated, however, we might gain performance by avoiding the storage of the unreduced a. Ideally we will put this transformation as high as possible in the AST where we can while still preserving program's semantics, hopeful that forcing the evaluation of the expression sooner will be faster than relying on the order of evaluation set by the runtime system. For instance, in a term t_1 of the form $e_1 \ldots f' \ldots e_n$, we might want to force the evaluation of a' even earlier in the program execution by rewriting our original expression:

$$\begin{aligned} t_1' = \ & \textbf{let}\ a' = (1 + 2 + 3 + 4) \\ & \textbf{in}\ \ seq\ a'\ (e_1 \ldots (\lambda c \rightarrow f\ a'\ 5\ c) \ldots e_n) \end{aligned}$$

We also have cases in which some code might be "probably" required but not "provably" required and where early evaluation might still be beneficial even if its value is never used simply because it reduces memory use sooner. This depends on the probability of it being called. Also, the semantics of the program should not change even in cases of evaluation when code is not

needed, so it should not have side effects or diverge when the program otherwise would not. However, detecting such cases can easily become an undecidable problem.

In general, we have a compiler capable of transforming our input program into a mix of lazy and non-lazy code. Defining the best mix for all possible settings might not be possible, but as long as we only perform safe transformations, the output should still be correct. Through the use of a static analysis called *strictness analysis* we aim to detect a large sub-set of those optimization opportunities and use that information to improve our transformations.

2.1 Further examples

In order to make our presentation clearer, we first define a set of basic examples that can be used to quickly verify the behaviour of our analysis in some common cases. Most papers on previous work, shown on Chapter 3, also tend to rely on some of these examples to present their systems. Even in cases where the features presented in the code are not fully supported, it is still interesting for us to keep in mind these examples both to verify if the analysis still leads to safe transformations and to see if they affect other parts of the program.

$$
\begin{aligned}
&id :: a \to a \\
&id\ a = a \\
&twice :: (a \to a) \to a \to a \\
&twice\ a = a \circ a \\
&twiceId :: a \to a \\
&twiceId = twice\ id
\end{aligned}
$$

These functions allow us to test simple functions and function composition. Also, analysing $twice$ shows quite early how high order functions are treated in any given system.

$$
\begin{aligned}
&head :: [a] \to a \\
&head\ (x : xs) = x \\
&length :: Num\ a \Rightarrow [b] \to a \\
&length\ (x : xs) = 1 + length\ xs \\
&length\ [\,] \qquad = 0
\end{aligned}
$$

Both $head$ and $length$ allow us to verify how pattern matches behave and also how a basic recursive datatype like list is handled. Furthermore, $head$ also gives us an example of incomplete patterns.

$$
\begin{aligned}
&letY1 :: Num\ a \Rightarrow a \to a \\
&letY1\ y = \mathbf{let}\ f\ x = x + y
\end{aligned}
$$

```
            in  f y
letY2 :: Num a ⇒ a → a
letY2 y = let f x = x + y
          in  f `seq` 3
```

In *letY1* we have a **let** binding that makes use of the function in a fully saturated way, whereas in *letY2* it is not saturated, even if relevant. These two examples help us verify how information brought in by uses of these functions affect the strictness of the inner **let** expressions, if at all.

```
data MyEither a b = MyLeft ! a | MyRight b
fromMyLeft :: MyEither a b → a
fromMyLeft (MyLeft a) = a
fromMyRight :: MyEither a b → b
fromMyRight (MyRight b) = b
km x y z = if z ≡ 1
           then x − y
           else km y x (z − 1)
```

Through the *MyEither* datatype and its accompanying functions *fromMyLeft* and *fromMyRight* we can verify how our system supports user defined datatypes and how it handles strictness annotations on those. The function *km* tests basic use of recursion with complex flow of annotation information due to the switch of variable order in the recursive call. Both the datatype and the *km* examples were inspired by Glynn, Stuckey, and Sulzmann [3], although *MyEither* was called *Maybe* by those authors and had no forced strictness. That paper also contains a benchmark of their own code against regular compilation with the well established GHC compiler. This set of programs is in the **nofib** suite and we might also use it in order to validate our own system. It is available in the Git repository `http://darcs.haskell.org/nofib.git/`.

Chapter 3

Strictness Analysis

Before implementing any sort of transformation from lazy to strict code, we should define which analysis techniques will be used. Here we present a brief review of the main techniques, including the one already partially implemented inside UHC by Verburg [2]. We hope that doing so will help contextualize some of the analysis design choices and possible difficulties in supporting more advanced language features, such as higher order functions, data structures and recursion.

3.1 Abstract interpretation

Abstract interpretation is the representation of programs' semantics in an approximate way. We select a problem-specific domain $\mathfrak{D}^\sharp$ in which we can more easily verify or prove the safety, correctness, or other relevant characteristic for our program analysis. Through the use of some abstract semantic $\mathcal{S}^\sharp \in \mathbf{L} \to \mathfrak{D}^\sharp$ for our language $\mathbf{L}$, we transform programs to the domain we have selected and in which we can more easily verify our properties [4].

Mycroft's paper on transforming functions from a call-by-need call convention to call-by-value used an implementation of abstract interpretation [5]. In it, the author describes two different interpretations, $I^\sharp$ and I^b that are used to study cases in which the call convention transformation may be safely applied. $I^\sharp$ maps all terminating terms to 1 and some non-terminating terms to 0, whereas I^b maps all non-terminating terms to 0. Ultimately we only need to consider one interpretation to the domain $\mathfrak{D}^\sharp = \{0, 1\}$, where values are mapped to 0 when non-terminating and to 1 on all cases.

In this case, interpretation works by converting our input into formulas that allow us to check which arguments cause a function to diverge. We convert our functions to their abstract versions through $\mathcal{S}^\sharp$ and then iteratively check strictness on each of its arguments by setting the current argument to 0 and all others to 1.

For instance, we can pick the addition function, which is strict in both arguments. Indeed, we obtain the following interpretation and its evaluated parameter values:

$$
\begin{aligned}
&plus \;\; x\ y = x + y \\
&plus^\sharp\ x\ y = x \ \wedge\ y \\
&plus^\sharp\ 0\ 1 = 0 \ \wedge\ 1 = 0 \\
&plus^\sharp\ 1\ 0 = 1 \ \wedge\ 0 = 0
\end{aligned}
$$

The last two lines indicate that $plus$ is strict in both arguments.

We can also analyse control structures, like in the function if' below:

$$
\begin{aligned}
&if' \;\; p\ x\ y = \textbf{if}\ p\ \textbf{then}\ x\ \textbf{else}\ y \\
&if'^\sharp\ p\ x\ y = p \ \wedge\ (x \ \vee\ y) \\
&if'^\sharp\ 0\ 1\ 1 = 0 \ \wedge\ (1 \ \vee\ 1) = 0 \\
&if'^\sharp\ 1\ 0\ 1 = 1 \ \wedge\ (0 \ \vee\ 1) = 1 \\
&if'^\sharp\ 1\ 1\ 0 = 1 \ \wedge\ (1 \ \vee\ 0) = 1
\end{aligned}
$$

The last three lines tell us that if' is only strict in its first argument. While we know it will require one of x or y, we cannot give guarantees about the actual evaluation of either.

If our function contains further function calls, we can replace those with their abstract versions. That might not terminate, for instance, if we have recursive function definitions. It is possible to simply play it safe and consider the worst case, but we can also use some more elaborate strategies to generate more precise results.

For example, if we take the following function and its abstract interpretation:

$$
\begin{aligned}
&f \;\; x\ y\ z = \textbf{if}\ x \equiv 0\ \textbf{then}\ y * x\ \textbf{else}\ f\ (x - 1)\ z\ y \\
&f^\sharp\ x\ y\ z = x \ \wedge\ ((y \ \wedge\ x) \ \vee\ (f^\sharp\ (x \ \wedge\ 1)\ z\ y))
\end{aligned}
$$

We can calculate $f\ 0\ 1\ 1$ directly and verify that f is strict in x. However, defining any of y or z to 0 when $x = 1$ will lead to an equation that depends on the result of $f^\sharp$ itself. To solve this we can just approximate the value to 1, which should always be safe, but is not satisfactory. Alternatively, we can use a transformation $\mathbf{T}$ of values in the abstract interpretation to calculate the result of recursive calls in a fixed point manner. That is, we define such $\mathbf{T}$ to be our abstract

interpretation with 0 in its first iteration and with the result of the last previous iteration in the subsequent ones. We can then write our fixpoint iteration in the following manner:

$$\mathbf{T}^0\ f = 0$$
$$\mathbf{T}^1\ f = x\ \wedge\ ((y\ \wedge\ x)\ \vee\ (\mathbf{T}^0\ f)) = x\ \wedge\ ((y\ \wedge\ x)\ \vee\ 0) = x\ \wedge\ y$$
$$\mathbf{T}^2\ f = x\ \wedge\ ((y\ \wedge\ x)\ \vee\ (\mathbf{T}^1\ f)) = x\ \wedge\ ((y\ \wedge\ x)\ \vee\ (x\ \wedge\ y)) = x\ \wedge\ y$$

We have that $\mathbf{T}^1\ f \equiv \mathbf{T}^2\ f$, which is our fixed point. From that we can conclude that f is strict in y but not in z:

$$f\ 1\ 0\ 1 = (1\ \wedge\ 0) = 0$$
$$f\ 1\ 1\ 0 = (1\ \wedge\ 1) = 1$$

An actual implementation of the algorithm above has exponential complexity due to the formula comparison, which makes the fixpoint calculation expensive.

Further developments also added support for higher-order functions and lists to the abstract interpretation technique. For instance, Burn, Hankin, and Abramsky already described how one can lift our abstract domain to create a domain for higher-order functions [6]. In that lifted domain we create new bottom and top elements which we use instead of 0 and 1, respectively, as parameters in our abstract interpretation. And support for lists in abstract interpretation was described by Wadler through the use of a more complex abstract domain [7]. This domain includes not only bottom and top but also infinite lists and lists in which only some of its elements are bottom. By verifying both lists constructors, nil and cons, as functions against all possible cases we can perform the analysis. The author hints at the generality of this technique to other datatypes, such as trees or composed lists. However, no mechanism to automatically populate the abstract domain from datatype definitions has been formally presented, so user defined datatypes are not supported.

3.1.1 Abstract reduction

Despite its advantages, abstract interpretation can get complex quite quickly. Fixpoint analysis is computationally expensive and domains with finite size do not easily support extending language with new data-types. With that in mind, but still keeping some of its advantages and methods, Nöcker developed an analysis system called abstract reduction [8]. For it, we use an infinite domain that can accommodate new data-types easily and that can represent various forms of non-termination.

Similar to how the interpretation works, we also need an abstract domain for abstract reduction. We first define a set $\mathbf{S}$ of the abstract representation of all function and constructors in our

language and add a bottom (Bot), a top (Top), and a special union value. Bottom represents all non-terminating values, top represents all values, and union represents a set of several possible reductions, used when running the algorithm. From that large set we can define our domain as graphs over **S** that represent the possible reduction paths from the abstract representation of our program.

Once we have a representation of our program we can process the analysis by feeding bottom and top values per argument and observe the result, like we did in abstract interpretation. However, instead of merely evaluating the abstract functions, we use an algorithm to identify on which alternative patterns, or execution paths, of a function we should verify the variable. This algorithm analyzes the input program on these alternatives using bottom and top values. If one or more alternatives match, we take their union. Without recursion, which would get us into reduction cycles, this algorithm eventually leads us to one of the four possible matching values:

- TotalMatch: the pattern or path represents the input bottom/top value entirely;
- PartialMatch: a Top value for the pattern;
- BotMatch: a Bot value for the pattern;
- NoMatch: other cases.

These results tell us which patterns have to be analysed. For instance, PartialMatch indicates that further matches, or paths, need to be verified and added to the output union, but the current one need not, as it may or may not terminate. Through these results we get an abstract reduction that mimics the original program quite accurately.

Instead of using a fixpoint approach for dealing with recursion, the authors let the algorithm run until it reaches a hard set limit on memory use, time use, or on the number of reduction steps. Once that limit is reached, the terms with recursive calls are replaced by Bot if verified that it is still needed. Otherwise, it may always be replaced by Top, which is safe. Informally, we say that it is needed if the term is either in a strict position or in all alternatives of reduction. It might be necessary to have dependency analysis before applying the algorithm so we can check functions in the right order and have all necessary strictness information in time.

It is also worth noting that Nöcker mentions that high-order functions are supported by the reduction system as long as currying is available.

Arbitrary data-types can be easily supported merely by adding those constructor possibilities to the set of possible symbols in the domain. However, in some cases, we might do better by

including further elements to the domain. For instance, we might want to encode list head and tail strictness differently, along with potentially infinite lists. The downside is that this complicates the system, since now we have to encode more strictness possibilities directly in the system and verify up to 4 possibilities of values, themselves able to reduce even further recursively. This is similar to the treatment of lists in abstract interpretation.

3.1.2 Comportment analysis

Also somewhat similar to the systems mentioned above, Cousot and Cousot presented a system called comportment analysis [9]. It "generalizes strictness, termination, projection (including absence), dual projection (including totality) and PER analysis and is expressed in denotational style." This analysis uses a much more complex lattice than the ones presented so far but seems to be reasonably precise and complete. The authors are also able to prove equivalences, after simplification of lattices, to earlier work, such as Mycroft's strictness analysis.

Its target language is a basic simply typed lambda calculus with pairs, Booleans, conditionals and fixpoint. There is little to no information about the practicality and even usefulness of implementing the full system as presented in the article, or even on how to expand it to support user defined datatypes and other advanced language features. However, it seems to have a very solid mathematical foundation explaining the reasoning for each design choice, paving the way to a more complete system.

3.2 Projection

A projection is an idempotent function that may remove information from its argument. Different from abstract interpretation and reduction, it does not alter the original function's domain. Through the use of projections and projection transformers it is possible to implement strictness analysis by verifying which projections may be safely applied to our input functions [10], [11].

A function α is a projection when it is continuous, discards information, and does so only once. It never alters or adds information to the input. Formally, we say that α has to respect the following properties:

$$\begin{aligned} \alpha\, u &\sqsubseteq u \\ \alpha\,(\alpha\, u) &= \alpha\, u \end{aligned}$$

For instance, the identify function id is clearly a projection, which we call ID. This element is also top in any lattice of projections, as it is the function that removes the least amount of

information. Typically, a projection BOT that maps every input to $\perp$ is the bottom of that lattice.

Before we go on, it is important to explain some notation. Given a function f and projections α and β, f is β-strict in context α if $\alpha \circ f = \alpha \circ f \circ \beta$, and we write it as $f : \alpha \Rightarrow \beta$. This basically means that β will never remove more information than what α needs in f.

We also define a new value element $\lightning$, called abort. It is used to indicate values that are not acceptable, as the function needs more information than available. All functions are strict in $\lightning$, so $f\ \lightning = \lightning$. We also need to add a way of handling $\lightning$ when taking the join of elements, done through the use of the $\&$ operator:

$$u \,\&\, v = \begin{cases} u \sqcup\ v & \text{, if } u \neq \lightning \text{ and } v \neq \lightning \\ \lightning & \text{, otherwise} \end{cases}$$

We can then define a strictness analysis by using carefully selected projections and verifying how our input function behaves with regard to those. More concretely, we define a strictness projection STR that considers the value $\perp$ unacceptable and then we verify if $f : STR \Rightarrow STR$ is the case. Formally, we define STR as:

$$STR\ u = \begin{cases} \lightning & \text{, if } u \text{ is } \perp \text{ or } \lightning \\ u & \text{, if } \perp \sqsubset \text{u} \end{cases}$$

Projection properties force us to consider $STR\ \perp \sqsubseteq \perp$, so we must have $\lightning \sqsubset \perp$ and $\lightning$ becomes our new domain bottom. Also because of that, we have to define a bottom projection called $FAIL$, which maps every input to $\lightning$, to be used instead of BOT. In addition to those, the lattice also contains ABS, or absent projection, which maps everything except $\lightning$ to $\perp$.

While this notion of β-strictness can be generalized to multiple argument functions, describing it can quickly become awkward. In order to simplify notation, the original authors defined projection transformers, which are functions that transform a projection of a function to projections that can be safely applied individually to one of that function's arguments. So if we call f^i a projection transformer to the i-th argument of a function f of arity n, and $\beta_i = f^i \alpha$ for every i, $1 \leqslant i \leqslant n$, we have that:

$$\alpha\ (f\ u_1 \ldots u_n) \sqsubseteq f\ (\beta_1 u_1) \ldots (\beta_n u_n)$$

Just like with functions, we also have transformers for individual variables inside expressions. We typically denote those as the expression itself indexed by the variable the transformer relates

to. For instance, for a given strict $\alpha \neq FAIL$, we have the following basic expression projection transformers:

$$\begin{aligned} x^x\ \alpha &= \alpha \\ y^x\ \alpha &= ABS \\ k^x\ \alpha &= ABS, \text{ if } k \text{ is a constant} \\ (f\ e_1 \cdots e_n)^x\ \alpha &= e_1^x\ (f^1\ \alpha)\ \&\ \cdots\ \&\ e_n^x\ (f^n\ \alpha) \end{aligned}$$

With those, we can define the strict and non-strict parts of a transformer, $STR \sqcap \alpha$ and $STR \sqcup \alpha$ respectively.

Using these carefully designed projections, rules, and operators, we can identify on which arguments of a function is it strict and which it is not. This analysis support datatypes directly with rules for creating projection transformers that pattern match on constructors. Function calls and recursion are handled like in abstract interpretation. For function calls we use their calculated transformed projections and recursion is handled through fixpoint iteration. The first iteration is defined as $FAIL$ instead of 0, and the following iterations are calculated on top of that.

3.3 Totality analysis systems

As an alternative to detecting strictness, we can focus on termination of expressions through totality analysis. If a function is strict, we can evaluate its argument before performing the call. Alternatively, if the argument is known to result in a head normal form, we can safely evaluate it before performing the call. As both occasions effectively result in the same idea of converting safe cases of call-by-name to call-by-value, they can be used in a somewhat interchangeable way.

Several techniques have been developed for totality analysis, and most of those involve type inference with effects carrying totality information. Type inference is typically described in a declarative way as a deductive system with inference rules composed of premises and a conclusion. These rules indicate how to (re)build the type information of the original expression and the final judgement has the form $\Gamma \vdash e : \tau$, where Γ represents the environment containing information necessary to type the expression e with type τ. For more expressive systems, like the ones below, τ is usually expanded to carry annotation information on top of the usual types. Chapter 6 describes how these systems may look like in more depth.

3.3.1 Kuo & Mishra's type inference system

For instance, Kuo and Mishra presented a technique for detecting non-termination of evaluation [12]. It first focuses on defining termination with regard to a head normal form, that is, a term converges under head evaluation if it results in one of the head normal forms. Otherwise, we say it diverges. The entire system consists of a phase for type inference with constraints, and another for constraint solving.

In this system, types are defined in terms of ϕ and $\Box$ identifiers. Also, we gather and solve a set of constraints C during the algorithm's execution. The system uses the following identifiers:

σ	$\in$	$TyExp$	Type expression
τ	$\in$	Ty	Type
α	$\in$	$TyVar$	Type variables
ϕ	$\in$	$\{e \mid \text{e is divergent}\}$	Type for divergent expressions
$\Box$	$\in$	Exp	Type for all expressions
w	$\in$	$Coercions$	Coercion
C	$\in$	$\mathcal{P}(Coercions)$	Coercion set

From those we can define the required types:

$$\begin{aligned} \tau &::= \phi \mid \Box \mid \alpha \mid \tau \to \tau \\ w &::= \tau \subseteq \tau \\ \sigma &::= C, \tau \end{aligned}$$

Note that $\phi \subseteq \Box$, as every divergent term is a term.

In other words, the type returned by the inference rules is σ, composed of the gathered constraints and the expression's type. Once we have the expression type, the constraint solver verifies that we have achieved a consistent set of constraints and simplifies them using some rules.

The authors argue that separating the constraint solving from the type inference allows us to express more types. As an example, the function $\lambda f\ x \to f\ (f\ x)$ may have any of the following types:

- $(\phi \to \phi) \to \phi \to \phi$
- $(\Box \to \Box) \to \Box \to \Box$

- $(\square \to \phi) \to \square \to \phi$

These several types cannot be expressed solely through parametrized types with type variables. We need to express that the second and fourth elements are equal and have to be smaller than the first and third, also equal among themselves. Later works presented solutions that avoid this by allowing conjunctive types, which give us a set of possible types instead of a single one [3], [13]–[15]. For this system we simply represent the final type as $\{\beta \subseteq \alpha\}, (\alpha \to \beta) \to \alpha \to \beta$.

This system supports higher-order functions, but the authors did not describe how to properly integrate recursion and arbitrary datatypes.

3.3.2 Jensen's system with polymorphic support

Jensen's approach looks rather similar, but uses conjunctive types instead of constraints and uses a strictness logic to help simplify strictness properties [13]. This system also allows conditional and polymorphic strictness properties. With all those features, the author argues that it becomes equivalent to Burn, Hankin, and Abramsky's abstract interpretation.

Instead of $\square$ and ϕ, we now use the properties $\mathtt{t}$, true for all types, and $\mathtt{f}$, for undefined values. For instance, using the conjunctive type, $\lambda f\ x \to f\ (f\ x)$ could then be described as $(\mathtt{t} \to \mathtt{t}) \to \mathtt{t} \to \mathtt{t} \wedge (\mathtt{f} \to \mathtt{f}) \to \mathtt{f} \to \mathtt{f} \wedge (\mathtt{t} \to \mathtt{f}) \to \mathtt{t} \to \mathtt{f}$. Furthermore, operators for conditional strictness properties, $?$, and its dual, $\Leftarrow$, are introduced. They are defined as:

$$\varphi\ ?\ \alpha = \begin{cases} \mathtt{f} & \text{, if } \alpha \equiv \mathtt{f} \\ \varphi & \text{, otherwise} \end{cases} \qquad \varphi \Leftarrow \alpha = \begin{cases} \mathtt{t} & \text{, if } \alpha \equiv \mathtt{f} \\ \varphi & \text{, otherwise} \end{cases}$$

These allow expressions to be represented more precisely. For instance, we can type the **if** operator as $\varphi\ ?\ \varphi_1$, where φ_1 is the type for the conditional and φ is the type for the expressions. Meaning it will have strictness indicated by φ as long as the conditional can be evaluated. If the conditional does not terminate, neither does the evaluation of **if**.

This system supports recursion and higher-order functions, but polymorphic and algebraic datatypes were not introduced.

3.3.3 Strictness Meets Data Flow

In a way similar to Jensen above, Schrijvers and Mycroft presented a type and effect system with data flow information on effects [15]. It lacks the conditional strictness and polymorphic variables but has sequential composition of effects ($\phi_1 \cdot \phi_2$) and parameters with subeffecting constraints ($\phi <: \gamma$) to emulate both features. On top of those features, effects also contain branching ($\phi_1 + \phi_2$) and 0 as a minimal element.

Perhaps the most notable side effect of having the data flow present is that we gain further optimization possibilities. The authors argue that this type and effect system is capable of guiding not only traditional call-by-value transformations expected from strictness analyses, but also *implicational strictness*. This form of strictness appears when we know a particular thunk will always be forced and can remove the "is-computed" check before loading its value. Not only that, but we are able to do it in a interprocedural manner since data flow with argument evaluation order information is exposed to outside functions. Additionally, we are also able to use the data flow information on the effects to see which arguments might be used and to guide selective inlining to expose further strictness optimization possibilities or to remove absent parameters from function calls.

3.3.4 Gasser, Nielson & Nielson's strictness and totality analysis

In a similar fashion, Gasser, Nielson, and Nielson presented a system that treats both strictness and totality [14]. They also use conjunctive type systems and use annotation values related to the evaluation of expressions to Weak Head Normal Form (WHNF).

The possible annotation values are **b**, **n**, and $\top$. **b** works as bottom, and is used when the expression does not evaluate to WHNF, while **n** represent expressions that do. $\top$ can be used in any expression, including cases where we do not have further information.

A good chunk of Gasser, Nielson, and Nielson's work relates to describing how strictness and totality type annotations may be coerced to avoid writing redundant rules. They then prove that the analysis is sound and argue that their system cannot be compared to most of the previous ones, saying that sometimes they achieve better results, sometimes worse. Finally, it is worth noting that this system was presented with fixpoint support already, but no integration with datatypes, although a possible solution for that has been mentioned.

3.3.5 Strictness analysis with HORN constraints

Most of the previously shown systems were developed for specifically designed languages, usually omitting certain features such as datatype support or recursion. Glynn, Stuckey, and Sulzmann presented a system running in GHC and supporting the full Haskell 98 language [3]. Therefore, it has support for recursion, high-order functions and datatypes, though in the case of recursive datatypes, only the head element is analysed.

It is based on constraints of a strictness logic system with HORN propositional formulas with conjunctions and existential quantifications. They also argue their system is correct by showing the equivalence to that of Burn, Hankin, and Abramsky's abstract interpretation for first-order programs.

The system uses Kleene-Mycroft iterations to find the types and constraints for the fix operator, but an alternative following a similar method as that of Kuo and Mishra is also presented. This alternative is not as precise but is also correct and seems to be faster.

3.4 Relevance typing

Totality analysis did not focus on strictness but achieved similar results by treating arguments that were guaranteed to reduce to HNF as safe to be evaluated strictly. Similarly, we can try to detect which arguments are guaranteed to be used during the evaluation of a function to its head normal form. Those arguments are then said to be relevant and may be evaluated. Typically the relevance can be inferred through non-standard type inference, hence the name "relevance typing".

Relevance implies strictness, because if the evaluation of a relevant abstraction diverges, it is a safe approximation to consider it strict as well. However, there are strict functions that are not relevant. One interesting example is $\lambda x \rightarrow error$, which is strict but not relevant [1].

3.4.1 Wright's head neededness analysis

Wright presented a method that used type inference with enhanced types to implement a form of relevance analysis called *head neededness analysis* [16]. These types carry information on the reduction behaviour of functions with regards to the arguments it receives. We say that a function head needs its argument if every reduction path of that function to head normal form contains a

descendent of the argument. When that happens, we say that it has a type head needed, represented by $\Rightarrow$. If it is not needed, then we say it is constant and represent it with $\not\Rightarrow$. For instance, $(\lambda x \to x)\ M$ beta reduces to M, meaning that the identity function will reduce to its argument for any application, so we may give it a type $\alpha \Rightarrow \alpha$. Alternatively, $(\lambda x \to y)\ M$ beta reduces to y, so its type is $\alpha \not\Rightarrow \beta$.

Those types are insufficient to represent higher-order functions. The function $\lambda f \to f\ x$ has types $(\alpha \not\Rightarrow \beta) \Rightarrow \beta$ and $(\alpha \Rightarrow \beta) \Rightarrow \beta$. To solve this issue, the author included arrow types with variables, represented by $\to_n$, where n is a unique index among all variable arrow types. That function can then be typed as $(\alpha \to_1 \beta) \Rightarrow \beta$ or similar.

In some cases, the function's type might depend on more than one single arrow variable type. To be able to represent these functions, the system also extends the function type constructors with a Boolean algebra, considering any variables instantiated to $\not\Rightarrow$ as false and all instances of $\Rightarrow$ as true. We can take the case of function composition, for instance: $\lambda x \to (f \circ g)\ x$ has a final type that depends on both f, with type $\beta \to_1 \gamma$, and g, with type $\alpha \to_2 \beta$. In this case, we would have the final type $\alpha\ (\to_1 \wedge \to_2)\ \gamma$.

In situations where functions have more than one argument with the same base type, keeping the original type equality might not lead to the most general type, as they might have different head neededness. A similar reasoning applies to arrow variables themselves. To counter this issue, we add fresh arrow and type variables and constraints of the form $b_1 \leqslant b_2$. On those constraints, b_1 and b_2 can be any arrow expressions, including variable arrow expressions, and they indicate that b_1 is substitution equivalent to b_2. We say that b_2 is a renaming instance of b_1.

Through the use of type inference and constraints, we are able to get the most general type for expressions such as $\lambda x \to \lambda y \to f\ (g\ x)\ (g\ y)$. Suppose f and g have (regular) types $\gamma \to \gamma \to \beta$ and $\alpha \to \gamma$, respectively. A first attempt at typing our function would result in f and g having types $\gamma \to_2 \gamma \to_2 \beta$ and $\alpha \to_1 \gamma$, respectively, and the final type would be $\alpha\ (\to_1 \wedge \to_2)\ \alpha\ (\to_1 \wedge \to_3)\ \beta$. But that assumes both x and y have the same head neededness. Through the use of constraints and fresh identifiers, however, we can generate the more general type $\alpha'\ (\to'_1 \wedge \to_2)\ \alpha''\ (\to''_1 \wedge \to_3)\ \beta$, with constraints such as $\alpha \to_1 \gamma \leqslant \alpha' \to'_1 \gamma'$ and $\alpha \to_1 \gamma \leqslant \alpha'' \to''_1 \gamma''$. This allows x and y and their respective applications to g to differ in head neededness, if necessary, later during type inference.

This inference system also supports polymorphism and recursion through a fixpoint combinator. Furthermore, the author argues that the system has also been extended to support arbitrary user-defined algebraic datatypes.

3.4.2 Amtoft's minimal thunkification

Amtoft bases himself on the work of Wright to develop a system focused converting programs from call-by-name to call-by-value [17]. We can convert a lazy, call-by-name program to call-by-value by wrapping every expression in a thunk. This thunk is a function around the expression that expects a dummy argument. During execution, this argument is passed to the thunk, forcing its evaluation and unwrapping it. The challenge becomes to detect which expressions do not need to be "thunkified" and "dethunkified" to avoid wasting computation time on those operations. In effect, this is the same as detecting strictness.

Just like with Wright, this system also has two basic arrow types: $\rightarrow_0$ for strict applications and $\rightarrow_1$ for the general case. Constraints are also used and are of the form $\geqslant$, which is similar to $\leqslant$ presented by Wright, and $\gg$. These constraints distinguish between arguments on contravariant and covariant positions. Arrows in contravariant positions allow us to find a *least* assignment to arrows in covariant ones. The details of how to manipulate these constraints were not presented in the paper itself, however.

This system supports recursion but does not seem to support polymorphic **let**s or user defined datatypes. The system is also shown to be correct. Interestingly, the author uses two different types of recursion, bounded and unbounded, to help prove correctness theorems with regard to recursion.

3.4.3 Making "stricterness" more relevant

Holdermans and Hage focuses on the issue of implementing strictness analysis in languages that contain operators for forced strict application, such as Haskell's seq operator [1]. This is important because that sort of forced strictness is often used to handle performance issues, and practical, real world implementation of strictness analysis should be aware of them. They present an analysis on a language with regular and strict applications, abstractions, **if** cases, Booleans, natural numbers, and $error$. In that language, we can rewrite seq into $\lambda x \rightarrow \lambda y \rightarrow ((\lambda z \rightarrow y) \bullet x)$, where $\bullet$ represents strict application.

They first present a type-driven call-by-value transformation and show how the system gets either unsound and effective, or ineffective and sound when handling strict applications naively. With that in mind, they develop a type system that has annotations for both demand, represented by φ, and applicativeness, ψ.

Demand and applicativeness annotations are both of either S or L. We can think of those values as "strict" and "lazy", or "small" and "big", respectively, as that is the relation between them:

$S \sqsubset L$. In this type system, arrows are annotated with demands. $\overset{S}{\to}$ represent computations that produce relevant abstractions when evaluated to weak head normal form, whereas $\overset{L}{\to}$ represent terms that may or may not produce relevant abstractions. Also, types on both sides of arrows are annotated with ψ, with S representing that it is guaranteed to be applied to an argument and L, that it may or may not be applied to an argument. The final annotated type $\widehat{\tau}$ is of the form:

$$\widehat{\tau} \quad ::= \quad Bool \quad | \quad Nat \quad | \quad \widehat{\tau_1}^{\psi_1} \overset{\varphi}{\to} \widehat{\tau_2}^{\psi_2}$$

Additionally, the result of judgements have types of the form $\widehat{\tau}^{(\varphi,\psi)}$, always holding $\varphi \sqsubseteq \psi$.

This system handles an issue mostly ignored in previous work on strictness analysis and that is quite necessary for practical implementations. However, it is still not suitable for languages such as Haskell due to the lack of support for type polymorphism, recursion, and algebraic datatypes. At least for type polymorphism and algebraic datatypes, the authors argue that these features are orthogonal to the issue they focused on and should not be difficult to implement.

3.5 Summary

Understanding other lines of development may give us a notion on the complexity of the problem at hand and of issues we might face when developing support for more advanced language features. Preferably we will support as many features as UHC itself supports, though its architecture allows us to selectively skip some of the more advanced ones, if necessary.

Other people have already tried to solve the main issue of implementing strictness analysis in a practical sense in the Utrecht Haskell Compiler. Lokhorst described an implementation on *TyCore*, an intermediate typed language for UHC [18]. As far as I know, *TyCore* is still not fully implemented and any analysis performed at that stage would have less ordering flexibility versus one implemented in *Core*, for which more transformations have already been developed and implemented.

The most recent work was that of Verburg, which follows the ideas adopted by Holdermans and Hage with some differences, like the use of counters instead of S and L values for applicativeness annotations. These counters are called saturation and allow easier handling of partial applications. It still lacks support for some basic Haskell features, such as data structures and handles only lambda-lifted code. However, as it already has a system and code in UHC, it is an inspiration for this project.

Chapter 4

The Utrecht Haskell Compiler

The Utrecht Haskell Compiler (**UHC**) aims to be a full Haskell 98 compliant compiler and was designed to be extensible and easy to work with. It was announced in its current form on April of 2009 and has been under development ever since. Although small projects may use it as main compiler, its architecture makes it more suitable as a compiler for developing experimental language features [19].

In order to isolate which parts we need to work on, we should know how the compiler is organized internally. The following sections present the overall UHC architecture and the core language, which was used for implementing our analysis and transformations, in detail.

4.1 The UHC Architecture

A program compiled with UHC is transformed into several different intermediate languages, each of which may contain several internal transformations in itself. Compilation steps are presented in Figure 4.1. The two most basic forms of compilation currently well maintained are compilation to bytecode for an interpreter and regular compilation to executables. In all cases, Haskell code is parsed and then converted, in sequence, to Essential Haskell, Core, and GRIN. At the end of the pipeline, it is then converted to either BC or executable, optionally through Silly, depending on the selected backend. On Core, GRIN, and Silly there are also a quite a few internal transformations to simplify and optimize the code.

In more detail, we have the following languages involved in the compiler:

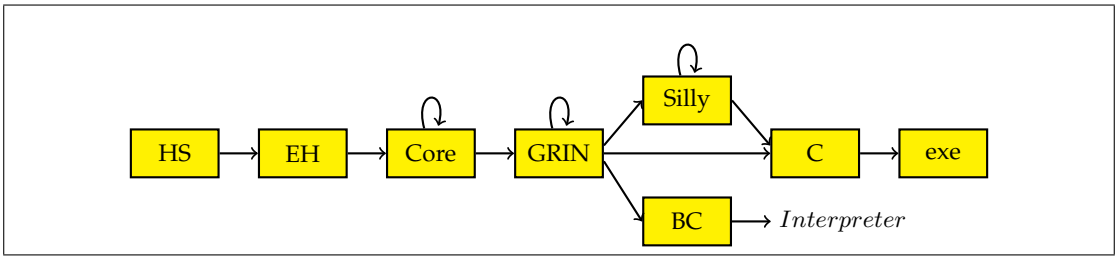

Figure 4.1: UHC pipeline for a single module.

- Regular **Haskell**, parsed from the source file.
- **Essential Haskell**, also called EH. It can be seen as a form of desugared Haskell. Name resolution, operator fixity and precedence, and name dependency are handled in this step.
- **Core** language is an untyped, lazy functional language, described more in detail in Section 4.2. Most, if not all, of this project will be implemented on this language.
- **GRIN**, or "Graph Reduction Intermediate Notation". It represents code in a "state monadic, first order, strict, functional program" with features to describe laziness. This language allows extracting better call graphs from lazy languages such as Haskell than other popular representations and is, therefore, well suited to be used before code generation [20].
- **Silly**, for "Simple Imperative Little Language" is a language that abstracts the imperative functionality of the possible target machines. It contains program control features, datatypes and primitives for stack manipulation that can be easily translated to low level languages such as C.
- **BC** is the bytecode for the UHC interpreter.

Boquist and Johnsson hinted at the possibility of implementing strictness analysis in GRIN directly [20]. However, it occurs very late in the pipeline and is mostly strict with lazy features. We believe that Core is better suited for this task as we may more easily experiment with different transformation orderings and the language is still lazy by default.

Before explaining Core itself, it is worth studying the use of attribute grammars and the aspect/variant concept used for incremental addition of language features in UHC.

4.1.1 Attribute grammars

The function *foldr* and other *fold*-like functions are good ways of abstract list traversal operations in functional programs. More generically, for any datatype, we can define one function

per constructor and apply those functions where appropriate while traversing it. This set of functions is called an *algebra* and they define a semantic of our data structure. The function that performs the traversal is called a *catamorphism*.

UUAG is an attribute grammar system that generates Haskell code from a specification containing the datatype and the desired catamorphism. This specification contains a description of the datatype we will operate on, the attributes we wish to generate, and semantic rules describing how those rules are generated. It is written in a language with features that make describing those catamorphisms much easier and less error prone. Details of that syntax are available in UUAG's user manual [21].

The description of datatypes follow mostly what is expected in Haskell, however every constructor field must be named so we may mention them directly in the semantic rules later. The following code describes in UUAG a datatype defined in plain Haskell as $\mathbf{data}\ Expr = EInt\ Int \mid Add\ Expr\ Expr$:

$$\begin{aligned}&\mathbf{data}\ Expr = EInt\ value :: Int\\ &\qquad\qquad \mid\ Add\ a :: Expr\ b :: Expr\end{aligned}$$

This informs UUAG of the datatype's constructors and how we may refer to each of the data fields later, when writing semantic rules. UUAG also has built-in support for some of Haskell's most common datatypes, such as $Maybe$ and $List$, and for type synonyms.

Attributes are values that may flow bottom-up, which we call synthesized, or top-down, called inherited. They may be summarized values, different trees or even just a transformed tree of the same type as input. An example would be to describe the result of the evaluation of the $Expr$ datatype above:

$$\mathbf{attr}\ Expr\ \mathbf{syn}\ result :: Int$$

This means we will have a synthesized attribute called $result$ of type Int.

Finally, semantic rules define how we want to generate the attributes. For generating the result, for instance, we may use the following rule:

$$\begin{aligned}&\mathbf{sem}\ Expr \mid EInt\ \mathbf{lhs}.result = @value\\ &\qquad\qquad \mid\ Add\ \ \mathbf{lhs}.result = @a.result + @b.result\end{aligned}$$

Here, the @ symbol indicates we are accessing some attribute or field from that particular constructor and is only necessary on the right hand side of the attribution. **lhs** indicates we are handling some data available on that constructor, in this case, $result$. We may omit some rules

on constructors by providing default functions and values. It is also possible to set some attribute properties, such as **copy**, which just propagates the original values up.

Compilers typically require a lot of parse tree traversals. In fact, each transformation between (and within) intermediate languages inside the pipeline as described above is a traversal in itself. In UHC, traversals are written in UUAG and takes advantage of all these features. Because of that, UHC's code is quite easy to work with.

4.1.2 Aspect-oriented architecture

In order to enable a simple incremental development approach in UHC, it was developed with *variants* and *aspects*. Variants are versions that incrementally add support for language features, starting from simply typed lambda calculus and ending in full Haskell. Aspects are features that are developed somewhat independently of variants. UHC's build system allows us to pick a variant and several aspects and generate a compiler with only those features.

Variants start at 1, with lambda-calculus and type checking and grow until 99 for all Haskell language features. Numbers 100 and 101 are typically used for testing and release, respectively. Intermediate numbers add support for datatypes, records, classes, modules, and other features.

Aspects represent features and compiler capabilities that may be enabled or disabled independently of variants, except for a possible minimal variant restriction. The two most common aspects are *typing* and *codegen*, responsible for typing and code generation, respectively. This built-in feature also allows us to isolate our strictness analysis in an aspect of itself, so it may be enabled without affecting other parts of UHC.

In order to handle all these different variants and aspects, UHC uses a tool called *shuffle*. It is responsible for taking chunk files of attribute grammars (.cag) and haskell source (.chs) and a list of aspects and variants and generate an output from those files containing only the relevant chunks. Chunks are defined by headers that identify which variants and aspects they refer to. They may also contain other information such as aliases to be used later and lists of chunks that should be removed before it is inserted. This is useful when mixing variants or aspects as some functions or datatype definitions may have to be redefined or expanded in the presence of extra language features.

4.2 The Core Language

Core is the most important of UHC's intermediate languages for this project. It is based on lambda calculus with all the required features to support Haskell, some of which are only enabled in later variants. As code generation is only available from variant 8 onwards, this is the one we should focus on at the beginning. In simplified terms, it is defined as:

$$
\begin{array}{rcl}
\text{CModule} & ::= & Name\ CExpr \\
\text{CExpr} & ::= & Int \mid Char \mid String \mid Var\ Name \mid Tup\ Tag \\
& \mid & FFI\ CallConv\ ForeignEnt\ Ty \\
& \mid & Lam\ CBind\ CExpr \mid App\ CExpr\ CBound \\
& \mid & Case\ CExpr\ [CAlt]\ CExpr \mid Let\ Categ\ [CBind]\ CExpr \\
\text{CAlt} & ::= & CAlt\ CPat\ CExpr \\
\text{CPat} & ::= & Var\ Name \mid Int \mid Char \mid Con\ Tag\ [Fields] \mid BoolExpr\ CExpr \\
\text{CBind} & ::= & Bind\ Name\ [CBound] \\
\text{CBound} & ::= & BBind\ CMetas\ CExpr \mid ... \mid FFE\ CallConv\ ForeignEnt\ CExpr\ Ty
\end{array}
$$

The actual language is actually slightly more complex and includes debugging information, annotations, and other language features such as support for extensible records, depending on which variant and aspects were selected. UHC makes a distinction between Foreign Function Interface calls and exports. The former is represented using FFI, the latter through FFE (for Foreign Function Exports). FFE is only enabled on variants 90 and above, despite being included here. The exact abstract syntax is defined in the file `src/ehc/Core/AbsSyn.cag`, relative to UHC's root path.

The Core language is composed of a module, defined by a name and an expression. An expression may be an int, char, string, variable, tuples or constructors, a FFI call, abstractions, case distinction and let bindings. The top level expression is typically a sequence of recursive occurrences of let bindings defining the top level entities of the module in an appropriate dependency order.

The *Tup* constructor represents either a tuple or a datatype constructor, depending on the information on the *Tag* data. It is always fully saturated inside the AST, with the datatype constructor always wrapped in a function. This way we can have partial applications to datatype constructors without breaking this saturation constraint. With tuples, *Tag* does not indicate arity, as these are defined by the number of consecutive applications inside the wrapping function.

The case expression contains the expression to be pattern matched, alternatives, and a default expression. Alternatives are written in terms of $CAlt$, which contain patterns and expressions. These patterns contain, among other possibilities, constructor cases that refer to the same tags used in Tup.

Finally, Let bindings contain category information indicating whether they are regular, strict or recursive bindings. On regular and strict bindings we may have a simple binding if we want to, but mutually recursive let definitions require them all to be available on the same level and, for that reason, Let has a list of bindings.

4.2.1 Comparison to GHC Core

When reading UHC core, it is interesting to keep in mind how other Haskell compilers behave with the same input. The obvious choice for comparison is GHC Core, GHC's intermediate representation language, given that GHC is the *de facto* standard Haskell compiler. The current version of GHC Core is based on system F_C, a variation of system F that includes support for generalized algebraic datatypes (GADTs) and associated types. [22]

Despite their differences, it is still possible to compare the generated intermediate representation from both compilers to some extent, as long as we take their differences into account. Because of those roots, and contrary to UHC's Core, GHC's is typed and contains big lambdas, type applications and explicit casts, all of which can usually be safely ignored for our purposes. Also, all **case** matching is strict in the scrutinee and that behaviour is used instead of defining a strict let binding used in UHC [23]. In other cases, they behave quite alike. For instance, GHC also wraps datatype constructors in functions (however, its considerably more powerful optimizations might eventually eliminate them).

In general, we will analyse GHC's behaviour in parallel to our developments without explicitly mentioning this in our text unless explicitly necessary for clarity.

4.3 Current UHC status

Since its creation in 2009, UHC has been maintained mostly by Atze Dijkstra with occasional contributions by students in the form of projects. Some examples of previous projects are the addition of support for LLVM and Javascript back-ends and Lokhorst's work on *TyCore*, mentioned at the end of Chapter 3.

Despite containing a few interesting experimental language features, UHC in its current state is not feature complete enough yet to be able to compile itself and relies on GHC for this. Additionally, the experimental and short lived nature of some of its associated projects resulted in a code base of non-consistent quality, with quite a few incomplete or not well maintained features. For instance, there is an on going effort to abstract both *Core* and *TyCore* through an interface called *AbstractCore*, which should be used whenever possible.

Chapter 5

Approach

We implemented our analysis and designed transformations as tree traversals inside UHC. Core was used as both source and destination languages and the source files were be added to the directory `src/ehc/Core/Trf`, following UHC's conventions. Transformations and analysis were designed to be as independent as possible to be added to different files. This made the distinction between steps explicit and, hopefully, make the code easier to work with and more reusable.

We followed these steps, in order:

1. Clean up previous work: Prepare code that already works to receive the changes, without disrupting normal compilation, like cloning the repository, set up the build system to accept the new aspect, add skeleton code files, and add example files;
2. Build infrastructure: Develop a basic framework with stub functions handling all rules so that development may be done stepwise;
3. Write first attempt: Implement basic language features to see how things run and which plan changes have to be performed;
4. Test: Verify that our examples run and work and that no unexpected side effects happen when the analysis or transformations are activated;
5. Add further functionality: Language features that were skipped on step 3 can then be developed and tested.

Given that there were attempts of implementing strictness analysis already and the code is still available, steps 1 and 2 were composed mostly of code reviewing and expansion. We needed to make sure the available code was compatible with the changes implemented to the analysis system.

The actual work, particularly on the first attempt, depended on which features we wished to implement and support. Roughly speaking, we have three different dimensions in this project: language features, transformations, and analysis. Defining how precise the analysis will be, which language constructs we should support, and where and how the transformations should take place are decisions that are relatively independent.

As transformation are not the focus, they should be quite simple and straightforward and occur only after analysis has been performed. Analysis precision depends on the features we would like to implement and language features will be implemented incrementally.

5.1 Analysis

Initially the analysis was as simple and pessimistic as possible and as features were added and matured, we could increase precision. This means providing results that are as close to the best valid strictness for the analysed terms as possible. We often lose precision due to termination or performance considerations, or simply due to the way the algorithm works. Regardless of those issues, we would like our development to be monotonic in this dimension.

However, to avoid issues later, we should keep in mind some of the issues we may face and that might affect the development of the basic framework. For instance, we would like to have the analysis context sensitive so we may use it in non-lambda-lifted code. This gives us more flexibility on the order of transformations inside the UHC pipeline. Also, we aim to make our system polyvariant to better handle higher order functions. This contrasts with existing GHC and UHC monovariant approaches. GHC relies on aggressive inlining to achieve good results when handling higher order functions, but this feature is not yet available in UHC, making it particularly dependent on polyvariance for performance gains.

The work of Holdermans and Hage and, more recently, Verburg was used as a base for the analysis system. However, learning from past experience and incrementing the system, some major changes were implemented. The first being the already mentioned polyvariance for this system, along with the context handling to deal with non-lambda-lifted code. Also, Verburg utilized three different annotations instead of Holdermans and Hage's two. Of those three, two were saturation counters in order to deal with partial applications. However, only the difference between them was needed and, in order to simplify this system, we used the two annotation approach and relied on careful use of annotations in arrow types to represent partial applications.

5.2 Language features

To ensure the project has a usable result, we will have to support at least variant 8, when code generation is added. Support for further variants will be added later, depending on their complexity, implementation maturity, and availability of our time. As our code relies entirely on Core, the only syntax changes we should need to worry about when supporting further variants are on the `AbsSyn.cag` file mentioned earlier.

Also, in order to isolate the changes introduced by our project from the rest of the compiler's functionality, it will all be introduced in an aspect before the code is merged back to the master development trunk. This approach allows to both reduce the impact of this project on UHC and will allow later comparisons between code generation and execution with and without these optimizations.

We do not plan to support every feature of variant 8 immediately. Initially, only Int, $Char$, $String$, Var, Tup, Lam, and App were supported. This gave us a nice overview to identify major issues in the overall approach and test for simple bugs in the transformation code. This is equivalent to step 3 mentioned above.

After testing, we repeated the cycle of adding functionality and then testing to implement more of the language. In order, they were plain and strict Let, $Case$, and then recursive Let.

5.3 Validation

In order to validate our results, and as part of the testing processes during development, we also perform some basic benchmarking. This means checking for changes in code size and compilation times when comparing UHC compilation with and without our code enabled, as well as comparing how it behaves at different stages of the UHC pipeline. Additionally, we should also verify that enabling the analysis and transformations does not alter the meaning of any program.

Chapter 6

Analysis

In order to describe the actual analysis system we detail some notation and background for our system. Just like the system presented by Holdermans and Hage, we were also inspired by the sub-structural type systems presented by Walker [24] to define our analysis. We define our system using inference rules that allow us to generate the final judgment, of the form $\Gamma \vdash e : \tau$, just like mentioned in Section 3.3.

Our presentation begins with the usual underlying type system, which we simplify to contain only information that may be relevant to us. Later we expand these judgments to carry annotations and restrictions as our problem demands. When reading the rules, one should keep in mind that relevance information flows backwards, towards the root of the AST.

6.1 Underlying type system

We define an underlying type system as a basis for the analysis. It will be altered to carry the relevance and applicativeness annotations. It should be based on Haskell's type system as this is our source language, but does not need to support all of it due to desugaring and other simplifications. Our analysis handles code that has already been parsed and type checked by UHC, so we may safely assume all input is type correct and only contains expressions in the Core language. The types in our system are represented by the following grammar:

$$\begin{array}{lcl} \tau & ::= & () \mid \alpha \mid \tau \to \tau \mid D\ Tag\ \overline{\tau_n} \\ \sigma & ::= & \forall \alpha.\sigma \mid \tau \end{array}$$

Despite being based on Haskell, we do not need to express as much as a regular Haskell program can. For instance, class instance constraints are not necessary, as earlier transformations in the compiler pipeline take care of replacing functions with their instances. Furthermore, we only care about the function and datatype structure of types and distinguishing between different primitive types is not relevant for strictness. So all primitive values share the same type, (), called unit. For example, a function of type $(Int \to Bool) \to Int$ may have type $(() \to ()) \to ()$.

We represent datatypes with a structure called D. With each datatype we associate its Tag structure and how its type variables are instantiated. The former indicates which datatype it refers to, while the latter lists the current value of its type variables, if any. Just like with the core language, we also use this datatype to represent tuples. Also, all tuples share the same generic Tag but differ in the number and value of the instantiated type variables. Later we will add information carried on datatypes to handle strictness annotations on datatype constructors.

Our system is polymorphic, so we need to be able to carry type variables, represented by α. For that, we use type schemes, represented by σ, which may quantify over type variables its inner type may contain.

As expected, we define a series of ftv functions to retrive free type variables of type schemes and types:

$$\begin{array}{lcl} ftv_\sigma(\tau) & = & ftv_\tau(\tau) \\ ftv_\sigma(\forall\alpha.\sigma) & = & ftv_\sigma(\sigma) \setminus \alpha \\ ftv_\tau(()) & = & \emptyset \\ ftv_\tau(\alpha) & = & \{\alpha\} \\ ftv_\tau(\tau_1 \to \tau_2) & = & ftv_\tau(\tau_1) \cup ftv_\tau(\tau_2) \\ ftv_\tau(D\ Tag\ \overline{\tau_n}) & = & \bigcup_{i=1}^{n} ftv_\tau(\tau_i) \end{array}$$

6.1.1 Environments

An environment is a mapping of identifiers to type schemes. It can be empty or an extension of an existing environment, as in:

$$\Gamma \quad ::= \quad [\,] \;\mid\; \Gamma[x \mapsto \sigma]$$

We can also write the environment by listing its bindings: $[x_1 \mapsto \sigma_1, ..., x_n \mapsto \sigma_n]$ is a shorthand for $[\,]\ [x_1 \mapsto \sigma_1] \ldots [x_n \mapsto \sigma_n]$.

We define $\Gamma(x)$ as the rightmost type associated with x in the environment Γ and it is not defined for x not in the domain of Γ, written $Dom(\Gamma)$. The function $ftv_\Gamma(\Gamma)$ is the union of the ftv_σ of the rightmost type for all bound identifiers in the domain of Γ.

6.1.2 Basic language

We develop our project around the Core language, described in Section 4.2. However, in order to make the description of the type systems easier to work with, we describe them through a more abstract version of that language. The following language can easily be derived from Core and is at least as expressive:

e	$\in$	$Exprs$	Expression
p	$\in$	$Pats$	Pattern
v	$\in$	$\mathbb{N}, Char, String$	Value
x	$\in$	$Vars$	Variable identifier

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid Con\ Tag\ \overline{e_n} \mid FFI\ \tau \\
& \mid & \lambda x \to e \mid e\ x \mid \textbf{case}\ e\ \textbf{of}\ \overline{p_n \to e_n} \\
& \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{let!}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{letrec}\ \overline{x_n = e_n}\ \textbf{in}\ e \\
p & ::= & v \mid x \mid Con\ Tag\ \overline{p_n}
\end{array}
$$

where e corresponds to $CExpr$ and p represents $CPat$ from Core. Integers, characters, and strings are stored as thunks to expressions converting a string representation to the actual value, but their lowest level arguments are always evaluated and their type is always unit so a single value v represents all three cases. Application has been changed so that we always apply an expression to a variable. Calling convention to an underlying runtime system via foreign functions should not, at this point, be relevant, so FFI only carries its type. Also, **let** types have been split into three different expressions: regular (**let**), strict (**let!**), and recursive (**letrec**). Regular and strict **let**s only carry one binding at a time as lists of bindings can be converted to nested **let**s but, due to mutually recursive definitions, recursive **let**s must have a list of bindings. Finally, the default pattern in **case** does not have to be written explicitly, as it may be converted to a pattern with a fresh variable.

In UHC, Tups are the representation for both datatype constructors and tuples. They carry the same Tag mentioned in Section 6.1 which identifies both the datatype and which constructor for that datatype it represents, among other information. However, tuples are treated specially in that their arity is not explicitly defined. Instead, UHC relies on the fact that every use of Tup occurs fully saturated in the code. To represent Tup, we just use Con with a list of expressions instead. We still use the Tag element used in Tup as it contains the identifier for the constructor, its arity (if not a tuple) and type and relevance information for each its fields.

To simplify our examples later, we will often write multiple successive lambdas using a single λ symbol and infix application as in regular Haskell when it may help readability. Additionally,

we may use regular **let**s with lists of bindings to shorten our inference tree. We will also use the application of functions to expressions without first defining a **let** binding of the value. Given our typing rules, presented below, we believe it is clear this should not alter the result while shortening the display of inference trees significantly and improving readability.

6.1.3 Underlying type system proper

With this background defined, we define the rules shown on Figure 6.1. Most cases are straightforward, as they behave as expected in the regular type system, with the difference that every base type is forced to be $()$.

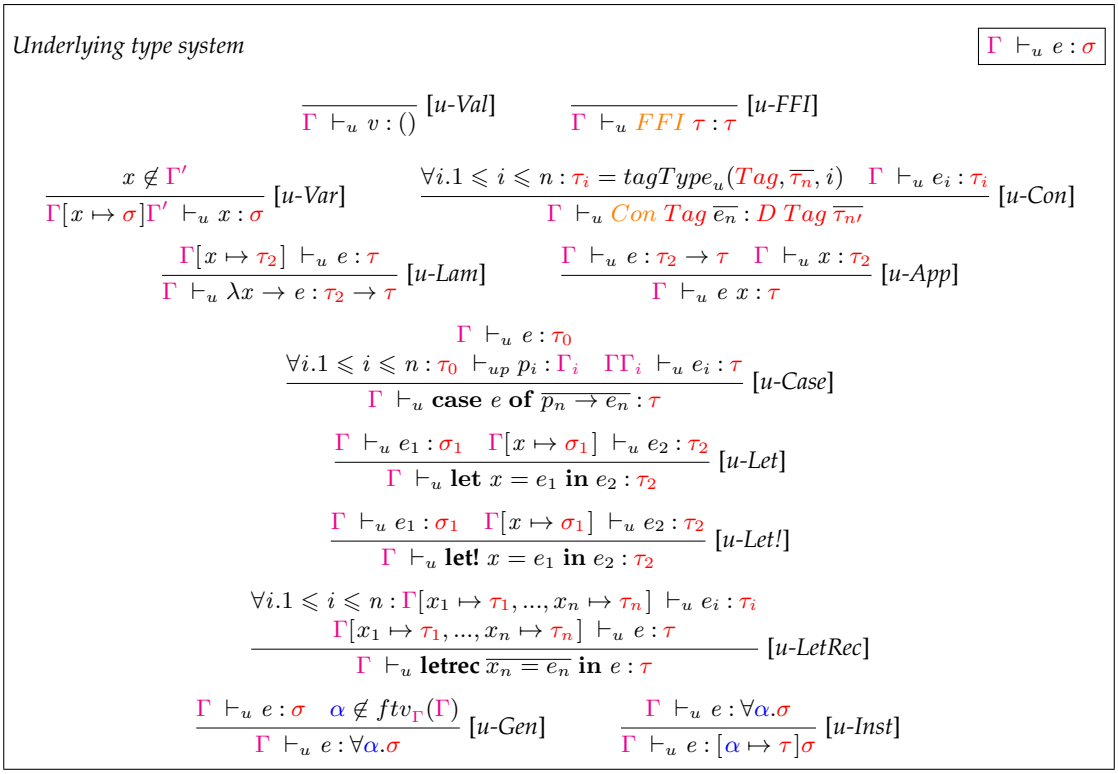

Figure 6.1: Underlying type system for the relevance analysis.

Rule [*u-FFI*] assumes the τ stored within FFI is valid and merely extracts it. We define an auxiliary function $tagType_u : Tag \to [\tau] \to \mathbb{N} \to \tau$ for [*u-Con*]. It takes the Tag, list of types and the index for the field selection that was stored within that constructor so we can properly infer its type.

Underlying type system for patterns $\boxed{\tau \vdash_{up} p : \Gamma}$

$$\frac{}{() \vdash_{up} v : []}\ [up\text{-}Val] \qquad \frac{}{\tau \vdash_{up} x : [x \mapsto \tau]}\ [up\text{-}Var]$$

$$\frac{\forall i.1 \leqslant i \leqslant n : \tau_i = tagType_u(Tag, \overline{\tau_{n\prime}}, i) \quad \tau_i \vdash_{up} p_i : \Gamma_i}{D\ Tag\ \overline{\tau_{n\prime}} \vdash_{up} Con\ Tag\ \overline{p_n} : \Gamma_1 \ldots \Gamma_n}\ [up\text{-}Con]$$

Figure 6.2: Type system for the pattern language, unannotated.

Rules for **case**s use an auxiliary environment Γ_i per pattern. These include the bindings each pattern introduces so we can properly type the inner expression e_i associated with it. To help with this we add a separate system, the underlying type system for patterns, defined in Figure 6.2.

There is no distinction between **let** and **let!** at this level as strictness does not affect types. Note that the type for the inner expression x is a σ instead of a τ. This fact, along with the treatment for σs in [*u-Var*] and the generalization and instantiation rules [*u-Gen*] and [*u-Inst*], allows us to represent types for polymorphic functions.

The [*u-LetRec*] rule does not describe in detail how to obtain the bindings as we only worry about having a consistent system. We also avoid the issue of typing polymorphic recursive functions by requiring the types for **letrec** to be τs.

6.2 Relevance for strictness

Holdermans and Hage developed their system by annotating types with demand and application information. We follow a similar path in that, in order to detect strictness, we focus on the uses of bindings in our environments, which we call relevance. We add annotations that indicate if an identifier is used or not inside the term being analyzed and store that information in the environment with its type. At the end of the analysis we have a set of accessible bindings, of which some are known to be relevant, per term of the program.

Relevance depends on both the binding and on the point of the program being analyzed, as the same binding may have different relevances at different points of the program. Our goal is to find the highest node in the AST in which a binding is relevant and transform the program to force its evaluation at that point. To aid in our goal, and also based on Holdermans and Hage, we use several auxiliary concepts, such as application annotations, containment and context splitting.

Before presenting our annotations and our annotated type system, we present a series of motivating examples to show, incrementally, why they are necessary and how they fit in.

6.2.1 Basic idea

Our first example is a simple addition function. Naturally, no one would have to write this, as any case where *f1* might be used, using $(+)$ would suffice. However, it is a nice example to show some of the basic concepts behind our system.

The input code is as follows:

$$f1 = \lambda x \to \lambda y \to x + y$$

We can clearly see that both x and y are used by the inner function and that it is safe to force the evaluation of y as soon as it is introduced. A naive approach may also do the same for x. However, that could be incorrect as we can only say at this point that x is relevant inside the inner lambda.

For instance, the following transformed *f1* is used in a relevant context, but x is not supposed to be evaluated because it will (incorrectly) cause an error in an otherwise clean terminating function:

$$\begin{aligned}&\textbf{let } f1' = \lambda x \to \textbf{let!} \ x' = x \\ &\qquad\qquad\quad \textbf{in} \quad \lambda y \to \textbf{let!} \ y' = y \ \textbf{in} \ x' + y' \\ &\textbf{in} \ \ f1' \ (error \ \texttt{"This fails."}) \ `seq` \ 1\end{aligned}$$

To avoid this issue, Holdermans and Hage use applicativeness and containment. Applicativeness propagates information that the function was used (or not used) back to its definition. If it is not guaranteed to be applied in full, then the applicativeness information that flows back to the definition of the lambdas triggers a containment of relevance. For instance, the binding for x found to be relevant inside the inner most abstraction will not be considered relevant between the lambdas and the transformation presented above is no longer valid.

Verburg [2] developed an alternative with a more complex applicativeness annotation to handle partial applications properly. It uses two saturation counters that keeps track of how many of the arguments of a function are passed and how many are still needed. We expand this idea to be able to deal with polymorphism and higher order functions.

6.2.2 More complex applicativeness

Given the previously defined notion of relevance, we developed our system to be as precise as possible. However, the addition of polymorphism, treatment for higher order functions, and support for user defined datatypes introduces some additional difficulties. Most of these issues can be observed easily from simple examples, as shown below.

We begin with a simple function defined in a **let** and used only once:

$$\begin{aligned} f2 = \lambda x \rightarrow \; & \textbf{let } f' = \lambda y \rightarrow x + y \\ & \textbf{in } \; f' \; 1 \end{aligned}$$

We can clearly see that x is relevant in f' because it is used inside its definition, in the first line. Additionally, as we can see that f' is used in a relevant context inside $f2$ because it is used in the body of the **let** binding, we know x will also be used and is also relevant inside $f2$, although indirectly.

One could see relevance as the only required attribute. However, as the example below shows, being in a relevant context is not enough:

$$\begin{aligned} f3 = \lambda x \rightarrow \; & \textbf{let } f' = \lambda y \rightarrow x + y \\ & \textbf{in } \; f' \; `seq` \; 1 \end{aligned}$$

Here, f' occurs in a relevant in the body of the **let** binding, but no arguments are passed to it. Therefore, f' is never reduced beyond WHNF and x's value is no longer needed, directly or indirectly, to calculate the result of $f3$. In this case, this means x should not be considered relevant in the body of the **let** binding or of $f3$.

On account of this, we add applicativeness information. We may try simply having one annotation of either "Fully applied" or "Other" (called $\top$, if in a lattice). This apparently solves some issues, but we then need to be able to properly define what it is to be fully applied. Take a higher order function, for instance:

$$\begin{aligned} const &= \lambda x \; y \rightarrow x \\ apply &= \lambda f \; x \rightarrow f \; x \\ f4 &= apply \; const \; 1 \end{aligned}$$

Here, $apply$ is fully applied, but the final type of the application in $f4$ is still a function. It is even worse that it returns a partially applied function that is derived from one of its original parameters. Clearly, this situation requires a more refined handling of applicativeness.

We initially tried to use a per argument applicativeness annotation. That is, for every argument, we propagate back the information on whether it had been applied or not. In $f4$ above we would be able to say that $apply$ has been fully applied, but $const$ has not.

This notion of applicativeness seemed to be sufficient for a while, but breaks easily with rather simple higher order functions. Say we have the following expression:

$$flip = \lambda f\ y\ x \rightarrow f\ x\ y$$
$$f5 = flip\ const\ 1$$

As we had decided to use annotated arrow types to carry application information, the type of *flip* could be $(y \xrightarrow{1} x \xrightarrow{2} z) \xrightarrow{3} x \xrightarrow{4} y \xrightarrow{5} z$, where numbers identify the application. If we are not careful, we may arrive at the conclusion that $const$'s second argument has been applied inside *flip* by making $1 = 5$ and $2 = 4$. This is obviously not the case, however, as no application takes place until *flip* is fully saturated.

One possibility is to always go for the safe assumption and consider them as not applied. This leads to an unacceptable loss of precision, given that higher order functions are heavily used in Haskell. To avoid that, we redesigned the applicativeness annotations to include information of related functions. We indicate at every arrow type, which other functions may also be considered (partially or fully) applied once the current one is.

In our new setting, every applicativeness annotation is followed by a set of related applicativeness identifiers. Once we receive an extra argument, the arrow identifiers and its related identifiers are considered applied. For instance, in the case above we could say that once we pass the argument for arrow 4, nothing else is considered applied. However, we consider the abstractions indicated by $\{1, 2\}$ to also be applied once arrow 5 receives its argument. We will no longer conclude that $const$ has been applied at all in $f5$ but we still maintain the information that we can reach that conclusion once an extra parameter is provided.

6.2.3 Polymorphism and polyvariance

One of the major differences between our system and that of Holdermans and Hage is that we have to support everything Haskell does in order to properly integrate our system in UHC. This means that polymorphism has to be handled. Verburg did so with a monovariant system. We consider a polyvariant version to greatly increase accuracy when handling polymorphism, although it also benefits us in some monomorphic functions.

Polyvariance is to annotations what polymorphism is to types. Whenever we bind a function we may also extend its annotations to depend on the instantiation we perform. As we use two types of annotations, it makes sense to consider universally quantifying on both annotations:

1. Relevance is a property local to the use of a binding. The main advantage in having polyvariant relevance is to be able to propagate relevance property through functions.

For instance, in $apply\ id$, the relevance for the final argument is instantiated to that of id, so we can conclude the final argument of $apply$ in this case will be relevant. Similarly, $apply\ (const\ 1)$ gets its final relevance instantiated to "not relevant", just like $const$'s second parameter.

2. Application is a property of the definition of the binding. Polyvariant application allows us to handle different higher order function parameters with precision.

A program that uses both $apply\ \ id$ and $apply\ \ (const\ \ 1)$ will be able to properly annotate the application of both id and $const$ (in its first argument only, so far) even if they are only applied through $apply$.

The code below presents an example on how polyvariance my increase precision:

$$
\begin{aligned}
f6 = \lambda y \rightarrow \mathbf{let}\ & apply = \lambda f\ x \rightarrow f\ x \\
& addy = \lambda x \rightarrow x + y \\
\mathbf{in}\ \ & apply\ addy\ 1
\end{aligned}
$$

As we apply $apply$ to $addy$, the "unleashed" annotations are that of $addy$ because $apply$'s first argument is applied once within its definition. We also observe that $apply$ is fully saturated and so the "unleashed" annotation tells us that $addy$ was also fully applied. Because of that, y may be considered relevant and forced to be evaluated before the body of the **let** and, in fact, before $f6$ is even called.

6.2.4 Context splitting

In some situations, we need to get information from two or more different branches of the AST and merge them. In the **case** below we can see these in action:

$$
\begin{aligned}
f6 = \lambda x\ y \rightarrow \mathbf{let}\ & f' = \lambda m \rightarrow m + y \\
\mathbf{in}\ \ & \mathbf{case}\ x\ \mathbf{of} \\
& \quad True\ \rightarrow f'\ 1 \\
& \quad False \rightarrow f'\ \text{`}seq\text{`}\ 1
\end{aligned}
$$

For bindings to be relevant, they need to occur in the scrutinee, because it is needed to pick the proper pattern, or in all patterns at once, so we can guarantee it will be needed. Our function $f6$ uses x in a relevant way, as it is the scrutinee of the **case** in the body of the **let**. However, y is

not relevant. It is used in f' and this, in its turn, is used on both case alternatives, but only fully applied in one of them.[1]

Context splitting is an operation that represents this notion that the bindings with the annotations for each sub term are somehow extracted from the final environment. That is, we "split" the final environment into two or more environments, all of which contain the same bindings, but possibly with differing annotations, to perform type inference for each of the sub terms. Another way of seeing this is that context splitting are guarantees that bindings are relevant or $\top$ in either or both sides, depending on the type of split.

Previous work used only one form of context splitting. We expanded this to two: one that allows joins and another that allows meets between annotations in the occurring bindings. For the **case** rule in *f6* above, for instance, we need both to represent that bindings are relevant if they are relevant on either the scrutinee or on both patterns.

6.3 Annotations

Holdermans and Hage used two variables to represent their demand and application annotations, φ and ψ, with two values each: S and L. Later, Verburg used this system as a basis but moved applicativeness to a pair of saturation counters. These counters keep track of how many arguments are required until the function can be reduced and how many have been applied so far. The original system system was developed in a rather limited language, with no datatypes, polymorphism or even **let** bindings. The system of Verburg supported most of UHC Core, but was still monovariant, as well as working only on lambda-lifted code.

We tackle the issue of strictness similarly, but in a polyvariant way and with some limited form of polymorphism support. Our focus is still on detecting which terms are relevant and at which nodes of the AST. All variables are considered relevant initially, but are changed to non-relevant (or rather, $\top$) at points where we can no longer guarantee they are still relevant. This may happen when joining or containing relevance annotations.

We annotate all types with relevance information and, in the case of functions, we also add relevance information per argument. The former propagate from the positions where they are used back to their definition in a backwards fashion, whereas the latter propagate from the definition of those functions to their uses. Relevance on types and on function arguments can be either R or $\top$, for *Relevant* and *Top*, respectively. These are equivalent to S and L as presented by Holdermans and Hage, but renamed to more closely represent their intended meaning. We use β to represent annotation variables, and φ_v to represent values.

[1] And these alternatives cover all cases for the Boolean datatype. But even if it did not, we could arguably ignore undefined alternatives as those diverge anyways. Diverging before a pattern match that would fail changes the message error, but not necessarily the expected behavior for the language.

$$\begin{array}{rcl} \varphi & ::= & \beta \mid \varphi_v \\ \varphi_v & ::= & R \mid \top \end{array}$$

Terms that have relevance annotation R are those that are guaranteed to have their evaluation demanded by the program on all possible execution paths from the point where that annotation is observed. If its annotation is $\top$, on the other hand, we are unable to provide such guarantee.

We have that $R \sqsubseteq \top$ or, more generally, for any φ, we have that $R \sqsubseteq \varphi$ and $\varphi \sqsubseteq \top$ hold. Similar to how Holdermans and Hage described a lattice with S and L, we say that $(\varphi, \sqsubseteq)$ forms a lattice with R and $\top$ as least and greater elements and joins and meets defined by:

$$\begin{array}{ll} R \sqcup \varphi = \varphi & R \sqcap \varphi = R \\ \top \sqcup \varphi = \top & \top \sqcap \varphi = \varphi \end{array}$$

The annotations per argument are written on arrows. It is used to indicate what is the expected behavior inside that function: when an arrow type has relevance annotation set to R, we know that that argument is going to be used. If its value is $\top$, on the other hand, it may be the case that it is not used, only used under certain circumstances, but not all, or we do not have enough information to say either way.

One of the main characteristics of the systems we base ours on is the notion of containment. It will be explained further in detail in Section 6.4.5 below, but for now, we present an intuition behind this concept. At every abstraction, we detect all relevances local to the body of that abstraction. We may only allow those relevances to "leak" to outside the lambda, however, if we detect that the function is applied in a relevant context. To do so while handling partial application, Verburg used two saturation counters as annotations on every variable. After some considerations, we decided to use a different approach in order to properly encode dependencies between applicativeness of higher order functions.

We first define a saturation, or applicativeness, annotation. It indicates whether, on all its execution paths, a particular arrow is fully applied. We call its values A and $\top$, short for *Applied* and *Top*, use γ as its variable, and ψ_v to represent values.

$$\begin{array}{rcl} \psi & ::= & \gamma \mid \psi_v \\ \psi_v & ::= & A \mid \top \end{array}$$

Just as with R and $\top$ above, they have the relation $A \sqsubseteq \top$, and $A \sqsubseteq \psi$ and $\psi \sqsubseteq \top$ hold for all ψ. Like before, $(\psi, \sqsubseteq)$ forms a lattice with A and $\top$ as least and greater elements and joins and meets defined by:

$$A \sqcup \psi = \psi \qquad A \sqcap \psi = A$$
$$\top \sqcup \psi = \top \qquad \top \sqcap \psi = \psi$$

On every arrow type we add a unique γ for identification purposes. We also add a set of related γs that are affected indirectly when the function of the current arrow is applied. That is, this related set indicates which other functions will be applied due to the fact that the current function was applied. We call γ sets ρ.

6.3.1 Annotated types and type environments

With these annotations defined can update our definitions for types and related concepts. For instance, our environment needs to map from identifiers to more complex type schemes, with annotations, and generalization over annotation variables. We call our new variants *hat* variants of our previously defined variables: $\widehat{\tau}$, $\widehat{\sigma}$ and $\widehat{\Gamma}$. We write $\widehat{\Gamma} \vdash e : \widehat{\sigma}\ ^{\varphi}$ for the judgment of an expression e under type environment $\widehat{\Gamma}$ with type $\widehat{\sigma}$, and relevance annotation φ.

Some of our changes are rather simple: references to τ and σ are replaced by $\widehat{\tau}$ and $\widehat{\sigma}$, respectively. We also add φ and ψ annotations, as well as related sets ρ to arrow types inside $\widehat{\tau}$, following the motivation described in Section 6.3 and add quantifications for annotation variables to type schemes. Finally, environments bind to annotated type schemes, allowing us to set the relevance for the entire binding. We define $\widehat{\tau}$ and $\widehat{\Gamma}$ as:

$$\begin{array}{lcl} \widehat{\tau} & ::= & () \mid \alpha \mid \widehat{\tau}\ \overset{(\varphi,\gamma,\rho)}{\rightarrow}\ \widehat{\tau} \mid \widehat{D}\ Tag\ \overline{\widehat{\tau}_n} \\ \widehat{\sigma} & ::= & \forall\alpha.\widehat{\sigma} \mid \forall\beta.\widehat{\sigma} \mid \forall\gamma.\widehat{\sigma} \mid \widehat{\tau} \\ \widehat{\Gamma} & ::= & [\,] \mid \widehat{\Gamma}[x \mapsto \widehat{\sigma}\ ^{\varphi}] \end{array}$$

We can also write lists of bindings inside $[\,]$ to describe annotated type environments, just like with regular type environments. We also define the operator $\setminus$ where $\widehat{\Gamma} \setminus x$ refers to the environment $\widehat{\Gamma}$ with all bindings for x removed. More generally, $\widehat{\Gamma}_1 \setminus \widehat{\Gamma}_2$ is $\widehat{\Gamma}_1$ removed of all variables that occur in $\widehat{\Gamma}_2$.

We bring special attention to the fact that ψ annotations are added to arrows as γ only. In practice, we only care if those identifiers occur in ρ (in which case they have been applied and are A) or not ($\top$). That is, for every γ, if $\gamma \in \rho$, then γ may be considered A. Otherwise, it is considered $\top$.

The ρ in arrows indicate γs of related arrows that should be considered applied whenever the arrow containing the ρ is applied. To see why this might be necessary, we may see the apply function written in two different ways:

The first is $\lambda f\ x\ y = f\ x\ y$, of type:

$$\forall \alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \gamma_1, \gamma_2. (\alpha_1 \xrightarrow{(\beta_1, \gamma_1, \emptyset)} \alpha_2 \xrightarrow{(\beta_2, \gamma_2, \emptyset)} \alpha_3) \xrightarrow{(R, \gamma_3, \emptyset)} \alpha_1 \xrightarrow{(\beta_1, \gamma_4, \emptyset)} \alpha_2 \xrightarrow{(\beta_2, \gamma_5, \{\gamma_1, \gamma_2\})} \alpha_3$$

Only once it has been fully applied we may consider the first argument, with applicativeness variables γ_1 and γ_2, to be also applied.

However, in the second version:

$$\begin{aligned} \lambda f\ x = &\ \textbf{let}\ f' = f\ x \\ &\ \textbf{in}\ \ \lambda y \to f'\ y \end{aligned}$$

It has type:

$$\forall \alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \gamma_1, \gamma_2. (\alpha_1 \xrightarrow{(\beta_1, \gamma_1, \emptyset)} \alpha_2 \xrightarrow{(\beta_2, \gamma_2, \emptyset)} \alpha_3) \xrightarrow{(R, \gamma_3, \emptyset)} \alpha_1 \xrightarrow{(\beta_1, \gamma_4, \{\gamma_1\})} \alpha_2 \xrightarrow{(\beta_2, \gamma_5, \{\gamma_2\})} \alpha_3$$

And here, if it has been applied up to the second argument, we may already consider the function f to be also partially applied up to its first argument.

Join and meet

Additionally, we define joins and meets for annotated types, type schemes and type environments. This allows us to handle large expressions without poisoning environments. Let $\square$ range over $\{\sqcap, \sqcup\}$, we define its application structurally:

$$\widehat{\tau} \begin{cases} () & \square & () & = () \\ \widehat{\tau}_1 \xrightarrow{(\varphi_1, \psi_1, \rho_1)} \widehat{\tau}_1' & \square & \widehat{\tau}_2 \xrightarrow{(\varphi_2, \psi_2, \rho_2)} \widehat{\tau}_2' & = (\widehat{\tau}_1 \square \widehat{\tau}_2) \xrightarrow{(\varphi_1 \square \varphi_2, \psi_1 \square \psi_2, \rho_1 \square \rho_2)} (\widehat{\tau}_1' \square \widehat{\tau}_2') \\ \widehat{D}\ Tag\ \overline{\widehat{\tau}_n} & \square & \widehat{D}\ Tag\ \overline{\widehat{\tau}_n'} & = \widehat{D}\ Tag\ (\widehat{\tau}_1 \square \widehat{\tau}_1' \ldots \widehat{\tau}_n \square \widehat{\tau}_n') \end{cases}$$

$$\widehat{\sigma} \begin{cases} \forall \alpha. \widehat{\sigma}_1 & \square & \forall \alpha. \widehat{\sigma}_2 & = \forall \alpha. \widehat{\sigma}_1 \square \widehat{\sigma}_2 \\ \forall \beta. \widehat{\sigma}_1 & \square & \forall \beta. \widehat{\sigma}_2 & = \forall \beta. \widehat{\sigma}_1 \square \widehat{\sigma}_2 \\ \forall \gamma. \widehat{\sigma}_1 & \square & \forall \gamma. \widehat{\sigma}_2 & = \forall \gamma. \widehat{\sigma}_1 \square \widehat{\sigma}_2 \end{cases}$$

$$\widehat{\Gamma} \begin{cases} [\,] & \square & [\,] & = [\,] \\ \widehat{\Gamma}_1[x \mapsto \widehat{\sigma}_1\ ^{\varphi_1}] & \square & \widehat{\Gamma}_2[x \mapsto \widehat{\sigma}_2\ ^{\varphi_2}] & = (\widehat{\Gamma}_1 \square \widehat{\Gamma}_2)[x \mapsto (\widehat{\sigma}_1 \square \widehat{\sigma}_2)\ ^{(\varphi_1 \square \varphi_2)}] \end{cases}$$

In the rules for $\widehat{\sigma}$ we assume both $\widehat{\sigma}_1$ and $\widehat{\sigma}_2$ are equal with regards to their quantified variables. That is, quantified αs, βs, and γs are the same on both sides.

As for γ sets, we define joins and meets as unions and intersections:

$$\rho_1 \sqcup \rho_2 = \rho_1 \cup \rho_2 \quad \rho_1 \sqcap \rho_2 = \rho_1 \cap \rho_2$$

Context splitting

Just like with Holdermans and Hage, we also define context splitting operations. Whereas they provided only one splitting operator, $\diamond$, we define two: $\diamond^{\sqcap}$ and $\diamond^{\sqcup}$. These allow us to represent that a particular term's annotation depends on the combination of two or more sub-analyses. The first operator represents the notion that the annotation is determined by at least one of the environments while the second represents that it is determined by its detected relevance on both environments.

Again, let $\square$ range over $\{\sqcap, \sqcup\}$, we define them as:

$$\begin{array}{rcll} [\,] & \diamond^{\square} & [\,] & = [\,] \\ \widehat{\Gamma}_1[x \mapsto \widehat{\sigma}^{\,\varphi_1}] & \diamond^{\square} & \widehat{\Gamma}_2[x \mapsto \widehat{\sigma}^{\,\varphi_2}] & = (\widehat{\Gamma}_1 \diamond^{\square} \widehat{\Gamma}_2)\,[x \mapsto \widehat{\sigma}^{\,(\varphi_1 \square \varphi_2)}] \end{array}$$

That is, we allow environments to be split into two copies as long as the resulting type annotations are compatible with the resulting annotations with the appropriate $\square$ operator. Aside from depending on this operator, we differ from previous work in that we define context splitting while allowing the presence of type quantifiers.

Free variables

As used in the underlying type system, we also have to define free variable functions for our annotated variables. However, this becomes a bit more involved since we also have to consider both annotation variables. Besides free type variables (ftv), we also define functions for free φ variables (fv^{φ}), and free ψ variables (fv^{ψ}). The functions ftv behaves mostly as defined earlier, except for the obvious change to the hatted variables and two extra cases in $\widehat{\sigma}$ to ignore β and γ variables.

The definition for ftv, fv^{φ}, and fv^{ψ} are given below.

$$
\begin{array}{lcl}
ftv_{\widehat{\sigma}}(\widehat{\tau}) & = & ftv_{\widehat{\tau}}(\widehat{\tau}) \\
ftv_{\widehat{\sigma}}(\forall\alpha.\widehat{\sigma}) & = & ftv_{\widehat{\sigma}}(\widehat{\sigma}) \setminus \alpha \\
ftv_{\widehat{\sigma}}(\forall\beta.\widehat{\sigma}) & = & ftv_{\widehat{\sigma}}(\widehat{\sigma}) \\
ftv_{\widehat{\sigma}}(\forall\gamma.\widehat{\sigma}) & = & ftv_{\widehat{\sigma}}(\widehat{\sigma}) \\
ftv_{\widehat{\tau}}(()) & = & \emptyset \\
ftv_{\widehat{\tau}}(\alpha) & = & \{\alpha\} \\
ftv_{\widehat{\tau}}(\widehat{\tau}_1 \rightarrow \widehat{\tau}_2) & = & ftv_{\widehat{\tau}}(\widehat{\tau}_1) \cup ftv_{\widehat{\tau}}(\widehat{\tau}_2) \\
ftv_{\widehat{\tau}}(\widehat{D}\ Tag\ \overline{\widehat{\tau}_n}) & = & \bigcup_{i=1}^{n} ftv_{\widehat{\tau}}(\widehat{\tau}_i)
\end{array}
$$

$$
\begin{array}{lcl}
fv^{\varphi}_{\widehat{\sigma}}(\widehat{\tau}) & = & fv^{\varphi}_{\widehat{\tau}}(\widehat{\tau}) \\
fv^{\varphi}_{\widehat{\sigma}}(\forall\alpha.\widehat{\sigma}) & = & fv^{\varphi}_{\widehat{\sigma}}(\widehat{\sigma}) \\
fv^{\varphi}_{\widehat{\sigma}}(\forall\beta.\widehat{\sigma}) & = & fv^{\varphi}_{\widehat{\sigma}}(\widehat{\sigma}) \setminus \beta \\
fv^{\varphi}_{\widehat{\sigma}}(\forall\gamma.\widehat{\sigma}) & = & fv^{\varphi}_{\widehat{\sigma}}(\widehat{\sigma}) \\
fv^{\varphi}_{\widehat{\tau}}(()) & = & \emptyset \\
fv^{\varphi}_{\widehat{\tau}}(\alpha) & = & \emptyset \\
fv^{\varphi}_{\widehat{\tau}}(\widehat{\tau}_1 \overset{(\varphi,\_,\_)}{\rightarrow} \widehat{\tau}_2) & = & \{\varphi\} \cup fv^{\varphi}_{\widehat{\tau}}(\widehat{\tau}_1) \cup fv^{\varphi}_{\widehat{\tau}}(\widehat{\tau}_2) \\
fv^{\varphi}_{\widehat{\tau}}(\widehat{D}\ Tag\ \overline{\widehat{\tau}_n}) & = & \bigcup_{i=1}^{n} fv^{\varphi}_{\widehat{\tau}}(\widehat{\tau}_i)
\end{array}
$$

$$
\begin{array}{lcl}
fv^{\psi}_{\widehat{\sigma}}(\widehat{\tau}) & = & fv^{\psi}_{\widehat{\tau}}(\widehat{\tau}) \\
fv^{\psi}_{\widehat{\sigma}}(\forall\alpha.\widehat{\sigma}) & = & fv^{\psi}_{\widehat{\sigma}}(\widehat{\sigma}) \\
fv^{\psi}_{\widehat{\sigma}}(\forall\beta.\widehat{\sigma}) & = & fv^{\psi}_{\widehat{\sigma}}(\widehat{\sigma}) \\
fv^{\psi}_{\widehat{\sigma}}(\forall\gamma.\widehat{\sigma}) & = & fv^{\psi}_{\widehat{\sigma}}(\widehat{\sigma}) \setminus \gamma \\
fv^{\psi}_{\widehat{\tau}}(()) & = & \emptyset \\
fv^{\psi}_{\widehat{\tau}}(\alpha) & = & \emptyset \\
fv^{\psi}_{\widehat{\tau}}(\widehat{\tau}_1 \overset{(\_,\gamma,\rho)}{\rightarrow} \widehat{\tau}_2) & = & \{\gamma\} \cup \rho \cup fv^{\psi}_{\widehat{\tau}}(\widehat{\tau}_1) \cup fv^{\psi}_{\widehat{\tau}}(\widehat{\tau}_2) \\
fv^{\psi}_{\widehat{\tau}}(\widehat{D}\ Tag\ \overline{\widehat{\tau}_n}) & = & \bigcup_{i=1}^{n} fv^{\psi}_{\widehat{\tau}}(\widehat{\tau}_i)
\end{array}
$$

As with ftv_{Γ} before, we define $ftv_{\widehat{\Gamma}}$ ($fv^{\varphi}_{\widehat{\Gamma}}$, $fv^{\psi}_{\widehat{\Gamma}}$) as the union of $ftv_{\widehat{\sigma}}$ ($fv^{\varphi}_{\widehat{\sigma}}$, $fv^{\psi}_{\widehat{\sigma}}$) for all bindings in $\widehat{\Gamma}$.

6.4 Relevance system

Given these definitions, we can alter the type system in Figure 6.1 and Figure 6.2 to include annotations. We need to be careful to maintain consistency throughout the system while still

being able to detect relevance whenever possible. That is, we do not add any rules that would allow the system to be inconsistent and we try to avoid simplifications whenever possible to get more precise results. On top of that, we still need to add rules to handle polymorphism, polyvariance, and subsumption, and any necessary auxiliary functions. All these definitions are given below.

6.4.1 Values

The first rule, for values, is quite straightforward:

$$\frac{}{\widehat{\Gamma}, \emptyset \vdash v : () \ ^{\varphi}} \ [a\text{-}Val]$$

Values may be considered relevant or not, so we do not specify any particular φ. As expected, the type is unit.

6.4.2 FFI

Foreign function interfaces carry their own types around and they are immutable. In order to have them consistent with our system, this type has to be annotated. As no further information is provided, we just annotate this carried type with safe annotation values, that is, set all annotations to $\top$.

To achieve all that, we define an auxiliary function, $annFFI : \tau \to \widehat{\tau}$:

$$annFFI(\tau) = \begin{cases} annFFI(\tau_1) \ ^{(\top,\top,\emptyset)}\!\!\to \ annFFI(\tau_2) & , \text{ if } \tau \equiv \tau_1 \to \tau_2 \\ () & , \text{ otherwise} \end{cases}$$

The rule using that function to type a FFI term becomes:

$$\frac{\widehat{\tau} = annFFI(\tau)}{\widehat{\Gamma}, \emptyset \vdash FFI \ \tau : \widehat{\tau} \ ^{\varphi}} \ [a\text{-}FFI]$$

The use of a FFI may still be relevant, even if the type for that FFI call is not. Because of this, we do not restrict the return type annotation and leave it represented by φ.

6.4.3 Variables

The rule for variables types a variable that must have a binding defined in the type environment. We also have to guarantee that, since the variable has appeared, it should be considered (locally) relevant. This is encoded by reading the binding inside the environment directly. Both this binding and the resulting type have to have the same annotations.

$$\frac{x \notin \widehat{\Gamma}'}{\widehat{\Gamma}[x \mapsto \widehat{\sigma}\ ^{R}]\widehat{\Gamma}', \emptyset \vdash x : \widehat{\sigma}\ ^{R}}\ [a\text{-}Var]$$

This way, the relevance annotations are shared between all uses of the same variable binding. Should x be a generalized type, we can expect instantiation rules to occur after [*a-Var*] in the derivation tree without affecting the binding relevance information.

6.4.4 Datatypes

Datatype constructors uses the same ideas we used for the unannotated version. We also use a $tagType$ function to retrieve the type for each field. However, we alter it so it will also return the strictness annotation value of the requested field, using the default $\top$ when it is absent. Its new type then becomes $tagType : Tag \to [\widehat{\tau}] \to \mathbb{N} \to \widehat{\tau}\ ^{\varphi}$. We then use that annotation and type when analyzing each field.

Note that constructors might not be in a context with enough type information for the datatype. The resulting type might contain variables that need to be quantified. However, this step is left for a later application of the generalization rules.

$$\frac{\forall i.1 \leqslant i \leqslant n : \widehat{\tau}_i\ ^{\varphi_i} = tagType(Tag, \overline{\widehat{\tau}_{n'}}, i) \quad \widehat{\Gamma}, \rho_i \vdash e_i : \widehat{\tau}_i\ ^{\varphi_i}}{\widehat{\Gamma}, \bigcup_{i=1}^{n} \rho_i \vdash Con\ Tag\ \overline{e_n} : \widehat{D}\ Tag\ \overline{\widehat{\tau}_{n'}}\ ^{\varphi}}\ [a\text{-}Con]$$

6.4.5 Abstractions

Much like in the original, unannotated system, we require a lambda term to be properly typed based on how its inner expression may be typed within the environment extended with a binding for the lambda variable. The issue arises when handling the propagation of annotations, whenever necessary.

Containment $\boxed{\varphi \blacktriangleright \widehat{\Gamma}}$

$$\frac{}{\varphi \blacktriangleright [\,]} \qquad \frac{R \blacktriangleright \widehat{\Gamma}}{R \blacktriangleright \widehat{\Gamma}[x \mapsto \widehat{\sigma}^{\,\varphi}]} \qquad \frac{\top \blacktriangleright \widehat{\Gamma}}{\top \blacktriangleright \widehat{\Gamma}[x \mapsto \widehat{\sigma}^{\,\top}]}$$

Figure 6.3: Containment rules for environments.

We make the φ_2 annotation from the variable x to be the arrow annotation in the final value. If the term is found to be relevant or not within the expression, then the arrow annotation should carry that information forward. We always consider its inner expression e to be relevant so its annotation is set to R to reflect this and allow strictness local to the expression to be detected.

At this point we should also be careful to contain demand annotations detected in the term e. That is, unless the function has been fully applied in a relevant context, we do not wish the environment $\widehat{\Gamma}$ to carry those relevances outside this lambda. To detect if the function has been applied, we can simply check if the local γ has been inserted into the ρ environment. To perform the containment of relevances, we use a containment operator $\blacktriangleright$, defined in Figure 6.3. It is similar to the ones used in previous work by Holdermans and Hage and works by allowing the environment to be populated with types annotated with safe relevance information with regard to the input demand and applicativeness annotations.

Using all these ideas, the rules for dealing with lambda abstractions becomes:

$$\frac{\top \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_2{}^{\varphi_2}], \rho \cup \rho_2 \vdash e : \widehat{\tau}^{\,R}}{\widehat{\Gamma}, \rho \vdash \lambda x \to e : \widehat{\tau}_2 \overset{(\varphi_2, \gamma, \rho_2)}{\to} \widehat{\tau}^{\,\varphi}} \; [\textit{a-Lam}]$$

$$\frac{\gamma \in \rho \quad \varphi \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_2{}^{\varphi_2}], \rho \cup \rho_2 \vdash e : \widehat{\tau}^{\,R}}{\widehat{\Gamma}, \rho \vdash \lambda x \to e : \widehat{\tau}_2 \overset{(\varphi_2, \gamma, \rho_2)}{\to} \widehat{\tau}^{\,\varphi}} \; [\textit{a-LamApp}]$$

6.4.6 Application

During application we have to match the arrow strictness with the argument. The other annotations are used in the expression e. In the conclusion we combine both environments used to type e and x. We use the operator $\diamond^{\sqcap}$ to do so, as it suffices for any variable to be relevant in one sub-expression of an application for it to be relevant for the entire expression.

Type system for patterns $\boxed{\widehat{\sigma}\ \vdash_p p : \widehat{\Gamma}}$

$$\frac{}{() \vdash_p v : []}\ [p\text{-}Val] \qquad \frac{}{\widehat{\tau}^{\,\varphi} \vdash_p x : [x \mapsto \widehat{\tau}^{\,\varphi}]}\ [p\text{-}Var]$$

$$\frac{\forall i.1 \leqslant i \leqslant n : \widehat{\tau}_i^{\,\varphi_i} = tagType(Tag, \overline{\widehat{\tau}_{n'}}, i) \quad \widehat{\tau}_i^{\,\varphi_i} \vdash_p p_i : \widehat{\Gamma}_i}{\widehat{D}\ Tag\ \overline{\widehat{\tau}_{n'}}^{\,\varphi} \vdash_p Con\ Tag\ \overline{p_n} : \widehat{\Gamma}_1 \ldots \widehat{\Gamma}_n}\ [p\text{-}Con]$$

Figure 6.4: Type system for the pattern language, annotated.

$$\frac{\widehat{\Gamma}_1, \rho_1 \vdash e : \widehat{\tau}_2 \overset{(\varphi_2, \gamma_2, \rho)}{\to} \widehat{\tau}^{\,\varphi} \quad \widehat{\Gamma}_2, \rho_2 \vdash x : \widehat{\tau}_2^{\varphi_2}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2, \rho \cup \{\gamma_2\} \cup \rho_1 \cup \rho_2 \vdash e\ x : \widehat{\tau}^{\,\varphi}}\ [a\text{-}App]$$

We also add the arrow application identifier γ_2 to the ρ set. This is the only way γs may be inserted into ρs, while the rule [*a-LamApp*] above removes it from the ρ set, which has to be empty at every leaf node. Having a single point of insertion and a single point of removal gives us the guarantee that no non-applied function is considered applied incorrectly.

6.4.7 Case

The **case** rule begins with type checking the scrutinee. This should always be done in a strict setting, so its relevance is set to R. Also, every pattern expression is analyzed locally as relevant so we may find local strictness information, similar to how we do in abstractions. Instead of using containment, however, we use context splitting in the conclusion to guarantee that only expressions that are relevant in all patterns or in the scrutinee may be considered relevant inside this expression.

We also define an auxiliary typing system, just like we did in the previous system. It is called $\vdash_p$ and is defined in Figure 6.4. With it, we generate the necessary bindings from each pattern which are then used to type e_i in an environment called $\widehat{\Gamma}'_i$. For each alternative we also get a split of the original environment, which we call $\widehat{\Gamma}_i$. The concatenation of $\widehat{\Gamma}_i$ and $\widehat{\Gamma}'_i$ is then used to analyze the expression.

$$\frac{\widehat{\Gamma}, \rho \vdash e : \widehat{\tau}_0^{\,R} \quad \forall i.1 \leqslant i \leqslant n : \widehat{\tau}_0^{\,\varphi} \vdash_p p_i : \widehat{\Gamma}'_i \quad \widehat{\Gamma}_i \widehat{\Gamma}'_i, \rho_i \vdash e_i : \widehat{\tau}^{\,R}}{\widehat{\Gamma} \diamond^{\sqcap} (\widehat{\Gamma}_1 \diamond^{\sqcup} \ldots \diamond^{\sqcup} \widehat{\Gamma}_n), \rho \cup \left(\bigcap_{i=1}^{n} \rho_i\right) \vdash \mathbf{case}\ e\ \mathbf{of}\ \overline{p_n \to e_n} : \widehat{\tau}^{\,\varphi}}\ [a\text{-}Case]$$

6.4.8 Let bindings

The previously defined rules did not touch three important Haskell aspects: polymorphism, strict application and recursion. We introduce them by carefully designing annotated rules for the three different let cases.

Regular let

The first let, the simplest one, introduces polymorphism. Just like in our unannotated version, we also need to add instantiation and generalization rules. As expected, we use the generalization rule when adding a new binding. To accommodate that, we use a $\widehat{\sigma}$ instead of specifying an annotated $\widehat{\tau}$ explicitly as the type for e_1. Otherwise we would not be able to have polymorphic and polyvariant types for x. Later, the type for every occurrence of x inside e_2 is instantiated as expected.

$$\frac{\widehat{\Gamma}, \rho_1 \vdash e_1 : \widehat{\sigma}_1{}^{\varphi_1} \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1{}^{\varphi_1}], \rho_1 \cup \rho_2 \vdash e_2 : \widehat{\tau}_2{}^{\varphi}}{\widehat{\Gamma}, \rho_2 \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \widehat{\tau}_2{}^{\varphi}} \text{ [a-Let]}$$

Since we need to represent three different types of quantifiable variables, we add three different generalization rules. All work similarly and differ only in the kind of variable and the use of the appropriate free variable function.

$$\frac{\widehat{\Gamma}, \rho \vdash e_1 : \widehat{\sigma}^{\varphi} \quad \alpha \notin ftv_{\widehat{\Gamma}}(\widehat{\Gamma})}{\widehat{\Gamma}, \rho \vdash e : \forall \alpha . \widehat{\sigma}^{\varphi}} \text{ [a-Gen]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e_1 : \widehat{\sigma}^{\varphi} \quad \beta \notin fv^{\varphi}_{\widehat{\Gamma}}(\widehat{\Gamma})}{\widehat{\Gamma}, \rho \vdash e : \forall \beta . \widehat{\sigma}^{\varphi}} \text{ [a-GenRel]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e_1 : \widehat{\sigma}^{\varphi} \quad \gamma \notin fv^{\psi}_{\widehat{\Gamma}}(\widehat{\Gamma})}{\widehat{\Gamma}, \rho \vdash e : \forall \gamma . \widehat{\sigma}^{\varphi}} \text{ [a-GenApp]}$$

Similarly, we have three different types of instantiation.

$$\frac{\widehat{\Gamma}, \rho \vdash e : \forall \alpha . \widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \vdash e : [\alpha \mapsto \widehat{\tau}]\widehat{\sigma}^{\varphi}} \text{ [a-Inst]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e : \forall \beta . \widehat{\sigma}^{\,\varphi}}{\widehat{\Gamma}, \rho \vdash e : [\beta \mapsto \varphi'] \widehat{\sigma}^{\,\varphi}} \text{ [a-InstRel]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e : \forall \gamma . \widehat{\sigma}^{\,\varphi}}{\widehat{\Gamma}, \rho \vdash e : [\gamma \mapsto \psi] \widehat{\sigma}^{\,\varphi}} \text{ [a-InstApp]}$$

The fresh $\widehat{\tau}$ that is instantiated is unified with other information obtained from analyzing the use of the polymorphic variable x inside the body of e for each of its uses. Similarly, the same happens to the fresh φ. They get instantiated to fresh names with unknown value and, depending on the necessary meets and joins with other variables and values, we are able to define safe, and hopefully accurate, values.

Strict let

For the strict let we use the same rule as the regular one with the difference that we have to always infer the relevances in e_1 in a relevant context, so we force it to be R. However, the binding for x may still have any relevance, as it can be unused or even used in non-relevant contexts inside e_2. The rule below indicates that.

$$\frac{\widehat{\Gamma}, \rho_1 \vdash e_1 : \widehat{\sigma_1}^{\,R} \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma_1}^{\,\varphi_1}], \rho_1 \cup \rho_2 \vdash e_2 : \widehat{\tau_2}^{\,\varphi}}{\widehat{\Gamma}, \rho_2 \vdash \textbf{let!}\ x = e_1\ \textbf{in}\ e_2 : \widehat{\tau_2}^{\,\varphi}} \text{ [a-Let!]}$$

Recursive let

Recursion, at this point, is let monovariant and monomorphic to simplify the rest of the rules for our system. Polyvariant recursion would require proving that the fixed point of the analysis can always be reached, overly complicating our system.

$$\frac{\begin{array}{c}\forall i . 1 \leqslant i \leqslant n : \widehat{\Gamma}[x_1 \mapsto \widehat{\tau}_1^{\,\varphi_1}, ..., x_n \mapsto \widehat{\tau}_n^{\,\varphi_n}], \rho_i \vdash e_i : \widehat{\tau}_i^{\,\varphi_i} \\ \widehat{\Gamma}[x_1 \mapsto \widehat{\tau}_1^{\,\varphi_1}, ..., x_n \mapsto \widehat{\tau}_n^{\,\varphi_n}], \rho \cup \left(\bigcup_{i=1}^{n} \rho_i \right) \vdash e : \widehat{\tau}^{\,\varphi}\end{array}}{\widehat{\Gamma}, \rho \vdash \textbf{letrec}\ \overline{x_n = e_n}\ \textbf{in}\ e : \widehat{\tau}^{\,\varphi}} \text{ [a-LetRec]}$$

6.4.9 Subsumption

One extra property is the subsumption rules for both relevance and applicativeness. These come directly from the idea that $\varphi \sqsubseteq \top$ and $\psi \sqsubseteq \top$, so replacing either variable with its respective $\top$ should be safe.

$$\frac{\widehat{\Gamma}, \rho \mapsto e : \widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \mapsto e : [\varphi' \mapsto \top](\widehat{\sigma}^{\varphi})} \text{ [a-SubRel]}$$

$$\frac{\widehat{\Gamma}, \rho \mapsto e : \widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \mapsto e : [\psi \mapsto \top]\widehat{\sigma}^{\varphi}} \text{ [a-SubApp]}$$

6.4.10 Overview

For a quick review of the entire system, we present all judgment rules at once in Figure 6.5.

Annotated type system $\boxed{\widehat{\Gamma} \vdash e : \widehat{\sigma}}$

$$\frac{}{\widehat{\Gamma}, \emptyset \vdash v : ()^{\varphi}} \text{ [a-Val]} \qquad \frac{\widehat{\tau} = annFFI(\tau)}{\widehat{\Gamma}, \emptyset \vdash FFI\ \tau : \widehat{\tau}^{\varphi}} \text{ [a-FFI]} \qquad \frac{x \notin \widehat{\Gamma}'}{\widehat{\Gamma}[x \mapsto \widehat{\sigma}^{R}]\widehat{\Gamma}', \emptyset \vdash x : \widehat{\sigma}^{R}} \text{ [a-Var]}$$

$$\frac{\forall i.1 \leqslant i \leqslant n : \widehat{\tau}_i^{\varphi_i} = tagType(Tag, \overline{\widehat{\tau}_{n'}}, i) \quad \widehat{\Gamma}, \rho_i \vdash e_i : \widehat{\tau}_i^{\varphi_i}}{\widehat{\Gamma}, \bigcup_{i=1}^{n} \rho_i \vdash Con\ Tag\ \overline{e_n} : \widehat{D}\ Tag\ \overline{\widehat{\tau}_{n'}}^{\varphi}} \text{ [a-Con]}$$

$$\frac{\top \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_2^{\varphi_2}], \rho \cup \rho_2 \vdash e : \widehat{\tau}^{R}}{\widehat{\Gamma}, \rho \vdash \lambda x \to e : \widehat{\tau}_2 \xrightarrow{(\varphi_2, \gamma, \rho_2)} \widehat{\tau}^{\varphi}} \text{ [a-Lam]}$$

$$\frac{\gamma \in \rho \quad \varphi \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_2^{\varphi_2}], \rho \cup \rho_2 \vdash e : \widehat{\tau}^{R}}{\widehat{\Gamma}, \rho \vdash \lambda x \to e : \widehat{\tau}_2 \xrightarrow{(\varphi_2, \gamma, \rho_2)} \widehat{\tau}^{\varphi}} \text{ [a-LamApp]}$$

$$\frac{\widehat{\Gamma}_1, \rho_1 \vdash e : \widehat{\tau}_2 \xrightarrow{(\varphi_2, \gamma_2, \rho)} \widehat{\tau}^{\varphi} \quad \widehat{\Gamma}_2, \rho_2 \vdash x : \widehat{\tau}_2^{\varphi_2}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2, \rho \cup \{\gamma_2\} \cup \rho_1 \cup \rho_2 \vdash e\ x : \widehat{\tau}^{\varphi}} \text{ [a-App]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e : \widehat{\tau}_0^{R} \quad \forall i.1 \leqslant i \leqslant n : \widehat{\tau}_0^{\varphi} \vdash_p p_i : \widehat{\Gamma}_i' \quad \widehat{\Gamma}_i\widehat{\Gamma}_i', \rho_i \vdash e_i : \widehat{\tau}^{R}}{\widehat{\Gamma} \diamond^{\sqcap} (\widehat{\Gamma}_1 \diamond^{\sqcup} \ldots \diamond^{\sqcup} \widehat{\Gamma}_n), \rho \cup \left(\bigcap_{i=1}^{n} \rho_i\right) \vdash \textbf{case}\ e\ \textbf{of}\ \overline{p_n \to e_n} : \widehat{\tau}^{\varphi}} \text{ [a-Case]}$$

$$\frac{\widehat{\Gamma}, \rho_1 \vdash e_1 : \widehat{\sigma}_1^{\varphi_1} \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1^{\varphi_1}], \rho_1 \cup \rho_2 \vdash e_2 : \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}, \rho_2 \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \widehat{\tau}_2^{\varphi}} \text{ [a-Let]}$$

$$\frac{\widehat{\Gamma}, \rho_1 \vdash e_1 : \widehat{\sigma}_1^{R} \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1^{\varphi_1}], \rho_1 \cup \rho_2 \vdash e_2 : \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}, \rho_2 \vdash \textbf{let!}\ x = e_1\ \textbf{in}\ e_2 : \widehat{\tau}_2^{\varphi}} \text{ [a-Let!]}$$

$$\frac{\begin{array}{c}\forall i.1 \leqslant i \leqslant n : \widehat{\Gamma}[x_1 \mapsto \widehat{\tau}_1^{\varphi_1}, \ldots, x_n \mapsto \widehat{\tau}_n^{\varphi_n}], \rho_i \vdash e_i : \widehat{\tau}_i^{\varphi_i} \\ \widehat{\Gamma}[x_1 \mapsto \widehat{\tau}_1^{\varphi_1}, \ldots, x_n \mapsto \widehat{\tau}_n^{\varphi_n}], \rho \cup \left(\bigcup_{i=1}^{n} \rho_i\right) \vdash e : \widehat{\tau}^{\varphi}\end{array}}{\widehat{\Gamma}, \rho \vdash \textbf{letrec}\ \overline{x_n = e_n}\ \textbf{in}\ e : \widehat{\tau}^{\varphi}} \text{ [a-LetRec]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e_1 : \widehat{\sigma}^{\varphi} \quad \alpha \notin ftv_{\widehat{\Gamma}}(\widehat{\Gamma})}{\widehat{\Gamma}, \rho \vdash e : \forall \alpha.\widehat{\sigma}^{\varphi}} \text{ [a-Gen]} \qquad \frac{\widehat{\Gamma}, \rho \vdash e : \forall \alpha.\widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \vdash e : [\alpha \mapsto \widehat{\tau}]\widehat{\sigma}^{\varphi}} \text{ [a-Inst]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e_1 : \widehat{\sigma}^{\varphi} \quad \beta \notin fv_{\widehat{\Gamma}}^{\varphi}(\widehat{\Gamma})}{\widehat{\Gamma}, \rho \vdash e : \forall \beta.\widehat{\sigma}^{\varphi}} \text{ [a-GenRel]} \qquad \frac{\widehat{\Gamma}, \rho \vdash e : \forall \beta.\widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \vdash e : [\beta \mapsto \varphi']\widehat{\sigma}^{\varphi}} \text{ [a-InstRel]}$$

$$\frac{\widehat{\Gamma}, \rho \vdash e_1 : \widehat{\sigma}^{\varphi} \quad \gamma \notin fv_{\widehat{\Gamma}}^{\psi}(\widehat{\Gamma})}{\widehat{\Gamma}, \rho \vdash e : \forall \gamma.\widehat{\sigma}^{\varphi}} \text{ [a-GenApp]} \qquad \frac{\widehat{\Gamma}, \rho \vdash e : \forall \gamma.\widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \vdash e : [\gamma \mapsto \psi]\widehat{\sigma}^{\varphi}} \text{ [a-InstApp]}$$

$$\frac{\widehat{\Gamma}, \rho \mapsto e : \widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \mapsto e : [\varphi' \mapsto \top](\widehat{\sigma}^{\varphi})} \text{ [a-SubRel]} \qquad \frac{\widehat{\Gamma}, \rho \mapsto e : \widehat{\sigma}^{\varphi}}{\widehat{\Gamma}, \rho \mapsto e : [\psi \mapsto \top]\widehat{\sigma}^{\varphi}} \text{ [a-SubApp]}$$

Figure 6.5: Annotated type system for the relevance analysis.

Chapter 7

Algorithm

Using the system presented in the previous chapter, we identified which values and variables are needed at each point in our program's syntax tree to calculate binding relevance. With that information we are able to construct an algorithm in a Haskell like language in a syntax directed way. This fits nicely into UHC's use of attribute grammars to describe transformations in Core, making the implementation rather straightforward. We also used the **W** type inference algorithm system as a starting point to make the implementation easier to write, and work with.

In this chapter, we will present the implementation of some of the simpler rules in Section 7.1. Section 7.2 describes the idea behind our unification algorithm for annotated types and how it differs from the simpler versions. Sections 7.3, 7.4, and 7.5 describe our handling of constructors, case pattern matching and let bindings, completing the support for our base language.

7.1 Simple rules

We consider simple rules to be those required for the basic definition of a generic lambda calculus based language. Therefore, it has to include variables, abstractions and applications. Given their relative simplicity, we also add values and FFIs to this list. The latter, despite being a rather complex language feature, has a rather simple and safe type inferred from the unannotated type provided by the Core language itself and should be simple to implement. Figure 7.1 shows rules for type inference of the aforementioned rules along with the type signature for the function performing the inference.

$$
\begin{array}{lll}
\mathsf{W} :: (\widehat{\Gamma}, \rho, \varphi, e) \to (\widehat{\tau}, \mathsf{Subst}, \rho, \widehat{\Gamma}) & & \\
\mathsf{W}(\widehat{\Gamma}, \rho, \varphi, v) & = ((), id, \emptyset, \widehat{\Gamma}) & \\
\mathsf{W}(\widehat{\Gamma}, \rho, \varphi, FFI\ \tau) & = \textbf{let}\ \widehat{\tau} = annFFI(\tau) & \\
 & \quad \textbf{in}\ (\widehat{\tau}, id, \emptyset, \widehat{\Gamma}) & \\
\mathsf{W}(\widehat{\Gamma}, \rho, \varphi, x) & = \textbf{let}\ \widehat{\sigma}\ ^{-} & = \widehat{\Gamma}(x) \\
 & \quad\ \ \theta_1 & = \{ x \mapsto \widehat{\sigma}\ ^{\varphi} \} \\
 & \quad\ \ (\widehat{\tau}, \theta_2) & = inst(\widehat{\sigma}) \\
 & \quad\ \ \theta & = \theta_2 \circ \theta_1 \\
 & \quad\ \ \widehat{\Gamma}' & = \theta\ \widehat{\Gamma} \\
 & \quad \textbf{in}\ (\widehat{\tau}, \theta, \emptyset, \widehat{\Gamma}') & \\
\mathsf{W}(\widehat{\Gamma}, \rho, \varphi, \lambda x \to e) & = \textbf{let}\ \alpha, \gamma_1\ \textbf{be fresh} & \\
 & \quad\ \ (\widehat{\tau}_2, \theta, \rho_2, \widehat{\Gamma}_2[x \mapsto \alpha^{\varphi_1}]) & = \mathsf{W}(\widehat{\Gamma}[x \mapsto \alpha^{\top}], \rho, R, e) \\
 & \quad\ \ \widehat{\Gamma}' & = \textbf{if}\ \gamma_1 \in \rho \\
 & & \quad \textbf{then}\ \varphi \blacktriangleright \widehat{\Gamma} \\
 & & \quad \textbf{else}\ \ \widehat{\Gamma} \\
 & \quad \textbf{in}\ (\theta\ \alpha\ ^{(\varphi_1, \gamma_1, \rho_2)}\!\!\to\ \widehat{\tau}_2, \theta, \emptyset, \widehat{\Gamma}') & \\
\mathsf{W}(\widehat{\Gamma}, \rho, \varphi, e\ x) & = \textbf{let}\ \alpha, \beta_1, \beta_2, \gamma\ \textbf{be fresh} & \\
 & \quad\ \ (\widehat{\tau}_1, \theta_1, \rho_1, \widehat{\Gamma}_1) & = \mathsf{W}(\widehat{\Gamma}, \rho, \varphi, e) \\
 & \quad\ \ (\widehat{\tau}_2, \theta_2, \rho_2, \widehat{\Gamma}_2) & = \mathsf{W}(\theta_1\ \widehat{\Gamma}, \rho, \beta_2, x) \\
 & \quad\ \ \theta_3 & = \{ \beta_2 \mapsto \beta_1 \} \\
 & \quad\ \ \theta_4 & = \mathsf{U}(\theta_2\ \widehat{\tau}_1, \widehat{\tau}_2\ ^{(\beta_1, \gamma, \emptyset)}\!\!\to\ \alpha) \\
 & \quad\ \ \theta & = \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \\
 & \quad\ \ \widehat{\Gamma}' & = (\theta\widehat{\Gamma}_1) \diamond^{\sqcup} (\theta\widehat{\Gamma}_2) \\
 & \quad\ \ \rho' & = \rho_1 \cup \rho_2 \cup \theta\{\gamma\} \\
 & \quad \textbf{in}\ (\theta\ \alpha, \theta, \rho', \widehat{\Gamma}') &
\end{array}
$$

Figure 7.1: Simple rules.

The function **W** takes an environment $\widehat{\Gamma}$, a set of applied arrows ρ, the local relevance φ and the expression to be analyzed. It returns the annotated type for the expression, a substitution, the set of applied arrows inside that expression and the environment containing updated relevance for the local bindings.

Rules for values and FFIs are straightforward, as their type is either simple (values) or simple to obtain from local information only (FFIs). They also return no transformation and no applied arrows, and do not alter the input environment.

Variables return empty ρs as they do not apply anything. However, they need to apply some transformations to the environment. The first is θ_1 which transforms the binding from the original type scheme and annotation to the same type scheme with the new relevance annotation. The second, θ_2, is calculated from the instantiation of the type scheme, if any. Both are combined and applied to the environment.

Abstractions need more changes than the previous rules, when compared to the basic **W** algorithm. First, we use the recursive call to infer the relevance of the inner expression e in a relevant context and pass the binding for x as non-relevant. If x is used in a relevant way or with a function whose relevance is a β variable, it will return with its binding annotation altered. We inspect the returned environment for that annotation and use that information to construct the final value. As we always add and remove bindings in an inverted order, we can use pattern matching for $\widehat{\Gamma}$ such as in the left hand side of the recursive call to remove extra bindings from environments. Any applied arrows or arrow identifiers within the body of the function are added as the return type's arrow ρ and any substitutions are propagated unchanged.

One major feature of the abstractions' rule is that we use a fresh γ_1 identifier for the newly generated arrow and immediately check to see if that identifier is in the set of applied arrows, the input ρ, before containing the environment. This assumes a top level circular propagation of applicativeness information: we first gather all return ρs from the full module's AST and only then use that information as the input for the call to the function **W**. It is somewhat similar to an expression such as $\mathbf{let}\ (x, y) = (y, 1)\ \mathbf{in}\ x$, where x depends on the value of y even if both are defined in the same tuple. This trick is possible due to the fact that there is no mutual dependency between returning and incoming ρs and that γ identifiers are globally unique.

Finally, we implement applications quite similarly to plain **W**: recursively infer the function and the argument and generate the substitution from the unification of both results. We also need to add a substitution for the relevance information of the argument to be equal to that of the unified type, and perform the context splitting between both updated environments. We also calculate the return set of applied arrows from both the function and the argument, and include the newly applied γ to that set, before returning it.

$$
\begin{aligned}
&\mathbf{U} :: (\widehat{\tau}, \widehat{\tau}) \to \mathsf{Subst} \\
&\mathbf{U}((), ()) = id \\
&\mathbf{U}(\alpha, \widehat{\tau}) = \{\alpha \mapsto \widehat{\tau}\} \\
&\mathbf{U}(\widehat{\tau}, \alpha) = \{\alpha \mapsto \widehat{\tau}\} \\
&\mathbf{U}(\widehat{\tau}_1 \overset{(\varphi_1,\gamma_1,\rho_1)}{\to} \widehat{\tau}_1', \widehat{\tau}_2 \overset{(\varphi_2,\gamma_2,\rho_2)}{\to} \widehat{\tau}_2') = \mathbf{U}(\widehat{\tau}_1, \widehat{\tau}_2) \circ \mathbf{U}(\widehat{\tau}_1', \widehat{\tau}_2') \circ \mathbf{U}_\varphi(\varphi_1, \varphi_2) \circ \mathbf{U}_\rho(\gamma_1, \rho_1, \gamma_2, \rho_2) \\
&\mathbf{U}(\widehat{D}\ Tag\ \overline{e_n}, \widehat{D}\ Tag\ \overline{e_n'}) = \mathbf{U}_s(\overline{e_n}, \overline{e_n'}) \\
&\mathbf{U}(\_, \_) = error\ \texttt{"Unification error."} \\
&\mathbf{U}_s :: ([\widehat{\tau}], [\widehat{\tau}]) \to \mathsf{Subst} \\
&\mathbf{U}_s([], []) = id \\
&\mathbf{U}_s(e_1 : \overline{e_1}, e_2 : \overline{e_2}) = \mathbf{U}(e_1, e_2) \circ \mathbf{U}_s(\overline{e_1}, \overline{e_2}) \\
&\mathbf{U}_s(\_, \_) = error\ \texttt{"Trying to unify datatype constructors of different arity."} \\
&\mathbf{U}_\varphi :: (\varphi, \varphi) \to \mathsf{Subst} \\
&\mathbf{U}_\varphi(\varphi_v, \varphi_v) = id \\
&\mathbf{U}_\varphi(\beta, \varphi) = \{\beta \mapsto \varphi\} \\
&\mathbf{U}_\varphi(\varphi, \beta) = \{\beta \mapsto \varphi\} \\
&\mathbf{U}_\varphi(\_, \_) = error\ \texttt{"Failed to unify phi."} \\
&\mathbf{U}_\rho :: (\gamma, \rho, \gamma, \rho) \to \mathsf{Subst} \\
&\mathbf{U}_\rho(\gamma_1, \rho_1, \gamma_2, \rho_2) = \{\gamma_1, \gamma_2 \mapsto (\gamma_1, \rho_1 \cup \rho_2)\}
\end{aligned}
$$

Figure 7.2: Unification algorithm.

7.2 Unification algorithm

The regular unification functions for the unannotated types cannot be used directly in our system, but they can provide guidance. Unification for unit types and type variables is straightforward. The first major changes come with arrows, where we also unify arrow relevance and applicativeness information, and datatype constructors, that can only be unified if they have the same Tag and arity.

Unification for relevance annotations is only defined for relevance variables. Relevance values can be ignored if they happen to be the same, but unifying different annotations results in error as it is not well defined. We could have defined unification between R with $\top$ or App and $\top$ to use subsumption, though guaranteeing soundness in these cases would require knowing any other kind of dependencies between annotations in our program.

At last, whenever unifying applicativeness information, we also unify the set of unleashed variables from both sides. The single line in the definition of $\mathbf{U}_\rho$ means "replace any γ_1 or γ_2 with γ_1 and set its accompanying ρ to the union of ρ_1 and ρ_2".

We note that, whenever combining several substitutions, we use composition $\circ$ and expect it to respect an application order from right to left, much like the regular function composition in Haskell.

$$
\begin{array}{lll}
\mathbf{W}(\widehat{\Gamma}, \rho, \varphi, Con\ Tag\ \overline{e_n}) = \textbf{let}\ x, \alpha, \overline{\alpha_{n'}}\ \textbf{be fresh} & & \\
\qquad e & = x\ \overline{e_n} & \\
\qquad \widehat{\tau} & = \widehat{D}\ Tag\ \overline{\alpha_{n'}} & \\
\qquad \overline{(\widehat{\tau}_n, \varphi_n)} & = tagTypes(\widehat{\tau}) & \\
\qquad \widehat{\tau}' & = \widehat{\tau}_1 \xrightarrow{(\alpha_1, \varphi_1, \emptyset)} \widehat{\tau}_2 \xrightarrow{(\alpha_2, \varphi_2, \emptyset)} \ldots \xrightarrow{(\alpha_{n-1}, \varphi_{n-1}, \emptyset)} \widehat{\tau}_n \xrightarrow{(\alpha_n, \varphi_n, \emptyset)} \widehat{\tau} & \\
\qquad (\widehat{\tau}'', \theta_1, \rho_1, \widehat{\Gamma}_1[x]) = \mathbf{W}(\widehat{\Gamma}[x \mapsto \widehat{\tau}'], \rho, \varphi, e) & & \\
\textbf{in}\ \ (\widehat{\tau}'', \theta_1, \rho_1, \widehat{\Gamma}_1) & &
\end{array}
$$

Figure 7.3: Algorithm for datatype constructors.

7.3 Datatype constructors

Our rule for datatype constructors uses the fact that they are always fully saturated within UHC. We create a new expression involving the application of an identifier with the expected $\widehat{D}$ type for that constructor to all field expressions. Any type or relevance information that may be obtained from the constructor is used at this point. Everything else is left as type or annotation variables to be inferred later.

7.4 Case and patten matching

Case pattern matching uses an auxiliary $\mathbf{W}_p$ function to correctly infer types for the list of patterns inside the **case** rule. At top level we perform the inference of the scrutinee e and use union for ρ and context splitting for the environments as expected.

The $\mathbf{W}_p$ function, however, works slightly differently. It takes all arguments $\mathbf{W}$ takes plus a $\widehat{\tau}$ indicating the expected type for the pattern. This is used to help generate the local binding environment for the pattern at each pattern, as they have to have the same type as the scrutinee.

Additionally, $\mathbf{W}_p$ mixes the information from all patterns using intersection of sets for ρ and $\diamond^{\sqcap}$ for context splitting. For this to work properly, however, the $[]$ case has to have the identity value for all patterns. This is why we constrain $\widehat{\Gamma}$ to $\top$ and use id and the full ρ value, as well as a fresh type variable. None of these elements should affect the output of our pattern matching.

$$
\begin{aligned}
&\mathsf{W}(\widehat{\Gamma}, \rho, \varphi, \textbf{case } e \textbf{ of } \overline{p_n \to e_n}) = \textbf{let } (\widehat{\tau}_0, \theta_0, \rho_0, \widehat{\Gamma}_0) = \mathsf{W}(\widehat{\Gamma}, \rho, \varphi, e) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\widehat{\tau}_1, \theta_1, \rho_1, \widehat{\Gamma}_1) = \mathsf{W}_p(\theta_0\ \widehat{\Gamma}, \rho, \widehat{\tau}_0, \varphi, \overline{p_n \to e_n}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \rho' \qquad\qquad\quad = \rho_0 \cup \rho_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \widehat{\Gamma}' \qquad\qquad\quad = (\theta_1\ \widehat{\Gamma}_0) \diamond^{\sqcup} \widehat{\Gamma}_1 \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in } (\widehat{\tau}_1, \theta_1 \circ \theta_0, \rho', \widehat{\Gamma}') \\
&\mathsf{W}_p :: (\widehat{\Gamma}, \rho, \widehat{\tau}, \varphi, \overline{p_n \to e_n}) \to (\widehat{\tau}, \mathsf{Subst}, \rho, \widehat{\Gamma}) \\
&\mathsf{W}_p(\widehat{\Gamma}, \rho, \widehat{\tau}, \varphi, [\,]) \qquad\qquad = \textbf{let } \alpha \textbf{ be fresh} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \widehat{\Gamma}' = \top \blacktriangleright \widehat{\Gamma} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (\alpha, id, \rho, \widehat{\Gamma}') \\
&\mathsf{W}_p(\widehat{\Gamma}, \rho, \widehat{\tau}, \varphi, (p \to e) : \overline{p_n \to e_n}) = \textbf{let } (\widehat{\tau}_1, \theta_1, \rho_1, \widehat{\Gamma}_1) \quad = \mathsf{W}_p(\widehat{\Gamma}, \rho, \widehat{\tau}, \varphi, \overline{p_n \to e_n}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \widehat{\Gamma}_2 \qquad\qquad\quad = binds(p, \widehat{\tau}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\widehat{\tau}_2, \rho_2, \theta_2, \widehat{\Gamma}'\widehat{\Gamma}'_2) = \mathsf{W}(\theta_1\ (\widehat{\Gamma}\widehat{\Gamma}_2), \rho, \varphi, e) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \theta_3 \qquad\qquad\quad = \mathsf{U}(\widehat{\tau}_1, \widehat{\tau}_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \rho_3 \qquad\qquad\quad = \rho_1 \cap \rho_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \widehat{\Gamma}_3 \qquad\qquad\quad = (\theta_2\ \widehat{\Gamma}_1) \diamond^{\sqcap} \widehat{\Gamma}' \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (\theta_3\ \widehat{\tau}_1, \theta_3 \circ \theta_2 \circ \theta_1, \rho_3, \widehat{\Gamma}_3)
\end{aligned}
$$

Figure 7.4: Algorithm for case and its support function.

7.5 Let bindings

Finally, we have to consider the three let binding cases. The simplest of them all is **let!**, as we know its binding to be relevant. Regular **let** requires some additional testing for relevance of the binding. Lastly, recursive **let**s require an auxiliary function to correctly handle the list of possible mutually recursive bindings.

In both **let!** and **let** we call W recursively on the binding and body expressions and use the result of the former in the argument of the latter after generalization, as expected. As we know the binding in **let!** is relevant, we can simply merge its ρ information in all cases. On the other hand, the binding in a **let** may or may not be relevant. We first inspect its relevance before deciding whether or not to join the ρ set from the binding expression.

In the **letrec** case we create an environment with bindings for all binding expressions and infer their types individually to generate a final substitution. This is then used to create a second environment of types for those bindings. We always assume them to have $\top$ relevance, though within each individual α we may still contain arrows with known R annotations. This relies on the regular inference of types used in the $\mathsf{W}_{\circlearrowleft}$ function.

$$
\begin{array}{lll}
\mathbf{W}(\widehat{\Gamma}, \rho, \varphi, \textbf{let } x = e_1 \textbf{ in } e_2) = \textbf{let } (\widehat{\tau}_1, \theta_1, \rho_1, \widehat{\Gamma}_1) & & = \mathbf{W}(\widehat{\Gamma}, \rho, R, e_1) \\
\qquad\qquad (\widehat{\tau}_2, \theta_2, \rho_2, \widehat{\Gamma}'[x \mapsto \_^{\varphi_1}]) & & = \mathbf{W}(\theta_1\ \widehat{\Gamma}[x \mapsto gen_{\theta_1\widehat{\Gamma}}(\widehat{\tau}_1)^{\top}], \rho, \varphi, e_2) \\
\qquad\qquad \rho' & & = \textbf{if } \varphi_1 \equiv R \\
& & \quad \textbf{then } \rho_1 \cup \rho_2 \\
& & \quad \textbf{else } \rho_2 \\
\qquad \textbf{in } (\widehat{\tau}_2, \theta_2 \circ \theta_1, \rho', \widehat{\Gamma}') & & \\
\mathbf{W}(\widehat{\Gamma}, \rho, \varphi, \textbf{let! } x = e_1 \textbf{ in } e_2) = \textbf{let } (\widehat{\tau}_1, \theta_1, \rho_1, \widehat{\Gamma}_1) & & = \mathbf{W}(\widehat{\Gamma}, \rho, R, e_1) \\
\qquad\qquad (\widehat{\tau}_2, \theta_2, \rho_2, \widehat{\Gamma}'[x]) & & = \mathbf{W}(\theta_1\ \widehat{\Gamma}[x \mapsto gen_{\theta_1\widehat{\Gamma}}(\widehat{\tau}_1)^{R}], \rho, \varphi, e_2) \\
\qquad\qquad \rho' & & = \rho_1 \cup \rho_2 \\
\qquad \textbf{in } (\widehat{\tau}_2, \theta_2 \circ \theta_1, \rho', \widehat{\Gamma}') & & \\
\mathbf{W}(\widehat{\Gamma}, \rho, \varphi, \textbf{letrec } \overline{x_n = e_n} \textbf{ in } e) = \textbf{let } \alpha_1, ..., \alpha_n, \beta_1, ..., \beta_n, \gamma_1, ..., \gamma_n \textbf{ be fresh} & & \\
\qquad\qquad \widehat{\Gamma}_1 = [x_1 \mapsto \alpha_1\ ^{(\beta_1, \gamma_1, \emptyset)}\!\!\rightarrow \alpha_1{}^{\top}, ..., x_n \mapsto \alpha_n\ ^{(\beta_n, \gamma_n, \emptyset)}\!\!\rightarrow \alpha_n{}^{\top}] & & \\
\qquad\qquad \theta_1 & & = \mathbf{W}_{\circlearrowleft}(\widehat{\Gamma}\widehat{\Gamma}_1, \overline{\alpha_n}, \overline{e_n}) \\
\qquad\qquad \widehat{\Gamma}_2 & & = gen_{\theta\widehat{\Gamma}}(\theta_1\ [x_1 \mapsto \alpha_1{}^{\top}, ..., x_n \mapsto \alpha_n{}^{\top}]) \\
\qquad\qquad (\widehat{\tau}_2, \theta_2, \rho_2, \widehat{\Gamma}'\widehat{\Gamma}'_2) & & = \mathbf{W}(\theta_1\ \widehat{\Gamma}\widehat{\Gamma}_2, \rho, \varphi, e_2) \\
\qquad \textbf{in } (\widehat{\tau}_2, \theta_2 \circ \theta_1, \rho_2, \widehat{\Gamma}') & &
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{W}_{\circlearrowleft} :: (\widehat{\Gamma}, \overline{\widehat{\tau}}, \overline{e}) \to \theta & & \\
\mathbf{W}_{\circlearrowleft}(\widehat{\Gamma}, [\,], [\,]) = id & & \\
\mathbf{W}_{\circlearrowleft}(\widehat{\Gamma}, \widehat{\tau} : \overline{\widehat{\tau}_n}, e : \overline{e_n}) = \textbf{let } \theta_1 & & = \mathbf{W}_{\circlearrowleft}(\widehat{\Gamma}, \overline{\widehat{\tau}_n}, \overline{e_n}) \\
\qquad\qquad (\widehat{\tau}', \_, \theta_2, \_, \_) & & = \mathbf{W}(\widehat{\Gamma}, \emptyset, \top, e) \\
\qquad\qquad \theta_3 & & = \mathbf{U}(\widehat{\tau}, \widehat{\tau}') \\
\qquad\qquad \textbf{in } (\theta_3 \circ \theta_2 \circ \theta_1) & &
\end{array}
$$

Figure 7.5: Algorithm for let cases.

Chapter 8

Transformation

The code presented in Chapter 7 merely encodes how to obtain the relevance information we need. It did not, however, indicate how that information can be used to direct a program transformation to actually enjoy the benefits of performing strictness analysis automatically. In this chapter we show how a simple program may be transformed using that information and how we expect such transformation to solve the issues presented in the motivation chapter.

First, we need to remember which bits of information are directly relevant to us from our algorithm. As this is a relevance analysis, our focus is on the relevance information of every binding. Additionally, bindings may contain varying relevance information at different points of the program. Fortunately, that information is already available in our implemented algorithm as the return $\widehat{\Gamma}$ value of every call to $\mathbf{W}$ and related functions, such as $\mathbf{W}_p$ and $\mathbf{W}_{\circlearrowleft}$. These contain not only the annotated types for every binding visible at that point in the program, but also what is its top level relevance.

Once we have gathered all the relevant bindings at every point in the program, we may start identifying at which points is strict application possible and useful. Naturally we could transform the program so that, at every point in the program, we wrap expressions into a **let!** containing alternative bindings for all relevant identifiers and then replacing all occurrences of these identified bindings by the newly introduced ones.

For instance, suppose we have an expression e whose return $\widehat{\Gamma}$ of our call to $\mathbf{W}([\,], \rho, R, e)$ included the binding $[x \mapsto \_^{\,R}]$. We can then replace e by the following code, where $e\ [x\ /\ x']$ represents e with all occurrences of x replaced by x'.

let! $x' = x$
in $e\,[x\,/\,x']$

We could repeat that transformation for every relevant binding in the local $\widehat{\Gamma}$, always using the result of the previous transformation as the input for the following.

However that would be quite inefficient, leading to both a slow transformation and a slow program. Instead, we can traverse the program and perform this transformation only on bindings that are known to not have been transformed before. This alone cuts off quite a lot of transformation time and avoids repeatedly checking for thunk evaluation status.

This is still a naive and inefficient approach. To better perform this transformation we should also take care to respect the dependencies between bindings so that no binding is forced to be evaluated before bindings it depends on were (and which are, due to the fact they are also called, also going to be relevant). However, this idea shows clearly how the previously described algorithms and judgment systems can provide us with the basic information to convert programs to their partially strict versions.

Chapter 9

Implementation and results

As stated in Chapter 5, we implemented our analysis in UHC as we developed it. This allowed us to both validate our design choices and observe complications that had to be dealt with. The code was rather compact for such a complex analysis, both due to UHC's extensive use of tools such as UUAG, and because similarities between our system and already implemented transformations inside allowed us to reuse a lot of the infrastructure in place.

In total we had slightly more than 1600 lines of code between the `EH.AnaDomain` module and its submodules, and the `EH.Core.Trf.AnaRelevance` module. Both were already present in the code, albeit disabled and only partially implemented. Additionally, most of the code was either adapted or removed so that our annotations could be used instead, although some snippets and design choices were kept due to their elegance and reusability. For instance, the instantiation code using higher order attribute grammars for recursively replacing variables was kept and even expanded to handle annotation variables as well. On the other hand, the representation of constraints between elements of the lattices was not used as our analysis kept constraint solving to a minimum thanks to the careful design of our judgment rules.

While developing the code we used our experience to improve the algorithm and the judgment rules presented in the earlier chapters. The basic cases proved to be relatively easy to work with, once the encoding of types was done. Additionally, our assumptions when dealing with theory proved to be valid in practice. For instance, due to the transformation of the code to A-Normal Form, all applications occurred between expressions and variables and handling the default pattern in **case** proved to be simple. On the other hand, handling **let**s were more complex than expected as we had to encode all three cases in a single rule, since UHC implements all three types of **let** bindings using the same constructor.

Due to the complexity of the theory behind our system and to the care needed to avoid repeating our predecessor's mistakes, such as overcomplicated code or oversimplified or inaccurate system, not enough time to fully encode our analysis was left. At the moment of this writing, some necessary bits to allow our code to be merged safely back to UHC's master were still missing or untested. The implementation for datatypes is complicated due to the large number of automatically derived functions for even the simplest example. Performance was also bad, particularly when handling datatypes, though this could be due to some yet untracked bugs. Aside from these issues and some yet undetected bugs, our implementation proved to work. Some small examples worked and we are confident, seeing that no major theoretical issues have come up, that once these problems are solved we should be able to complete the system and even enable it as a default UHC feature.

The code developed for this thesis, along with any posterior improvements can be found in the development fork of UHC at `https://github.com/passalaqua/uhc`. As of now the code is still isolated in the branch `polyvariant-strictness` and not yet merged back to master. The strictness analysis also needs to be enabled with `--optimise=StrictnessAnalysis` when running the compiler. And to observe its behavior, we recommend enabling debug by compiling variant 99 of UHC and also use the flag `--dump-core-stages=on` to see the results as a UHC Core file. The relevant file contains the module name `ana-relev`.

Chapter 10

Conclusion

The main objective of this project was the development of an accurate and complete strictness analysis for the Utrecht Haskell Compiler. This compiler has a lot of interesting features, such as its very modular, composable architecture that uses variants and aspects to isolate the development of language features, but it lacks in optimization transformations and currently generates rather unoptimized code. We attempted to solve this issue by focusing on detecting and avoiding unnecessary laziness through the use of static program analysis.

Our project was not the first to attempt to solve the presented issue, and not even the first to do it in the context of UHC. However, we our goals were more ambitious than our predecessors in that we aimed at higher accuracy through the use of polyvariance. We believed this would improve our handling of higher-order functions and polymorphism considerably over monomoprhic analyses while still being compatible with all Haskell features supported by UHC. And as we had the benefit of being able to learn from these preceding attempts at solving the same problem, we were able to go further.

Our results included judgment rules describing our ideas and code implementing these ideas inside UHC. Perhaps even surprisingly, we did not need to add too many simplifications that would force us lose precision to achieve our goals in our analysis and we had positive practical results indicating our theory is solid. Although more extensive testing and maturing is still needed before our code may be accepted to the master development branch of UHC or enabled by default, the quality of our preliminary results and the ease of development with UHC makes us confident it is merely a matter of time until we achieve this needed maturing.

Bibliography

[1] S. Holdermans and J. Hage, "Making "stricterness" more relevant", English, *Higher-Order and Symbolic Computation*, vol. 23, pp. 315–335, 3 2010, ISSN: 1388-3690. DOI: 10.1007/s10990-011-9079-7. [Online]. Available: http://dx.doi.org/10.1007/s10990-011-9079-7.

[2] G. Verburg, "Strictness Analysis in UHC", Master's thesis, Utrecht University, 2012.

[3] K. Glynn, P. Stuckey, and M. Sulzmann, "Effective strictness analysis with HORN constraints", English, in *Static Analysis*, ser. Lecture Notes in Computer Science, P. Cousot, Ed., vol. 2126, Springer Berlin Heidelberg, 2001, pp. 73–92, ISBN: 978-3-540-42314-0. DOI: 10.1007/3-540-47764-0_5. [Online]. Available: http://dx.doi.org/10.1007/3-540-47764-0_5.

[4] P. Cousot, "Abstract interpretation", *ACM Comput. Surv.*, vol. 28, no. 2, pp. 324–328, Jun. 1996, ISSN: 0360-0300. DOI: 10.1145/234528.234740. [Online]. Available: http://doi.acm.org/10.1145/234528.234740.

[5] A. Mycroft, "The theory and practice of transforming call-by-need into call-by-value", in *International Symposium on Programming*, ser. Lecture Notes in Computer Science, B. Robinet, Ed., vol. 83, Springer Berlin Heidelberg, 1980, pp. 269–281, ISBN: 978-3-540-09981-9. DOI: 10.1007/3-540-09981-6_19. [Online]. Available: http://dx.doi.org/10.1007/3-540-09981-6_19.

[6] G. L. Burn, C. Hankin, and S. Abramsky, "Strictness analysis for higher-order functions", *Science of Computer Programming*, vol. 7, pp. 249–278, 1986, ISSN: 0167-6423. DOI: 10.1016/0167-6423(86)90010-9. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167642386900109.

[7] P. Wadler, "Strictness analysis on non-flat domains", in *Abstract interpretation of declarative languages*, Ellis Horwood, 1987, pp. 266–275.

[8] E. Nöcker, "Strictness analysis using abstract reduction", in *Proceedings of the conference on Functional programming languages and computer architecture*, ser. FPCA '93, New York, NY, USA: ACM, 1993, pp. 255–265, ISBN: 0-89791-595-X. DOI: 10.1145/165180.165219. [Online]. Available: http://doi.acm.org/10.1145/165180.165219.

[9] P. Cousot and R. Cousot, "Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages)", in *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, IEEE, 1994, pp. 95–112. DOI: 10.1109/ICCL.1994.288389.

[10] P. Wadler and R. Hughes, "Projections for strictness analysis", in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science, G. Kahn, Ed., vol. 274, Springer Berlin Heidelberg, 1987, pp. 385–407, ISBN: 978-3-540-18317-4. DOI: 10.1007/3-540-18317-5_21. [Online]. Available: http://dx.doi.org/10.1007/3-540-18317-5%5C_21.

[11] K. Davis and P. Wadler, "Backwards strictness analysis: proved and improved", in *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, London, UK, UK: Springer-Verlag, 1990, pp. 12–30, ISBN: 3-540-19609-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=647555.729745.

[12] T.-M. Kuo and P. Mishra, "Strictness analysis: a new perspective based on type inference", in *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, ser. FPCA '89, New York, NY, USA: ACM, 1989, pp. 260–272, ISBN: 0-89791-328-0. DOI: 10.1145/99370.99390. [Online]. Available: http://doi.acm.org/10.1145/99370.99390.

[13] T. P. Jensen, "Inference of polymorphic and conditional strictness properties", in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98, New York, NY, USA: ACM, 1998, pp. 209–221, ISBN: 0-89791-979-3. DOI: 10.1145/268946.268964. [Online]. Available: http://doi.acm.org/10.1145/268946.268964.

[14] K. L. S. Gasser, H. R. Nielson, and F. Nielson, "Strictness and totality analysis", *Science of Computer Programming*, vol. 31, no. 1, pp. 113–145, 1998, <ce:title>Selected Papers of the First International Static Analysis Symposium</ce:title>, ISSN: 0167-6423. DOI: 10.1016/S0167-6423(96)00043-3. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642396000433.

[15] T. Schrijvers and A. Mycroft, "Strictness meets data flow", in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot and M. Martel, Eds., vol. 6337, Springer Berlin Heidelberg, 2011, pp. 439–454, ISBN: 978-3-642-15768-4. DOI: 10.1007/978-3-642-15769-1_27. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15769-1_27.

[16] D. Wright, "A new technique for strictness analysis", in *TAPSOFT '91*, ser. Lecture Notes in Computer Science, S. Abramsky and T. Maibaum, Eds., vol. 494, Springer Berlin Heidelberg, 1991, pp. 235–258, ISBN: 978-3-540-53981-0. DOI: 10.1007/3540539816_70. [Online]. Available: http://dx.doi.org/10.1007/3540539816_70.

[17] T. Amtoft, "Minimal thunkification", in *Static Analysis*, ser. Lecture Notes in Computer Science, P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, Eds., vol. 724, Springer Berlin Heidelberg, 1993, pp. 218–229, ISBN: 978-3-540-57264-0. DOI: 10.1007/3-540-57264-3_43. [Online]. Available: http://dx.doi.org/10.1007/3-540-57264-3_43.

[18] T. Lokhorst, "Strictness optimization in a typed intermediate language", Master's thesis, Utrecht University, 2010.

[19] A. Dijkstra, J. Fokker, and S. D. Swierstra, "The architecture of the Utrecht Haskell compiler", in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, ser. Haskell '09, Edinburgh, Scotland: ACM, 2009, pp. 93–104, ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596650. [Online]. Available: http://doi.acm.org/10.1145/1596638.1596650.

[20] U. Boquist and T. Johnsson, "The GRIN project: a highly optimising back end for lazy functional languages", in *Implementation of Functional Languages*, ser. Lecture Notes in Computer Science, W. Kluge, Ed., vol. 1268, Springer Berlin Heidelberg, 1997, pp. 58–84, ISBN: 978-3-540-63237-5. DOI: 10.1007/3-540-63237-9_19. [Online]. Available: http://dx.doi.org/10.1007/3-540-63237-9_19.

[21] A. Baars, D. Swierstra, and A. Löh, "UU AG system user manual", *Department of Computer Science, Utrecht University*, 2003.

[22] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly, "System F with type equality coercions", in *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, ser. TLDI '07, Nice, Nice, France: ACM, 2007, pp. 53–66, ISBN: 1-59593-393-X. DOI: 10.1145/1190315.1190324. [Online]. Available: http://doi.acm.org/10.1145/1190315.1190324.

[23] S. Peyton Jones and S. Marlow, "Secrets of the Glasgow Haskell Compiler inliner", *Journal of Functional Programming*, vol. 12, pp. 393–434, 4-5 Jul. 2002, ISSN: 1469-7653. DOI: 10.1017/S0956796802004331. [Online]. Available: http://journals.cambridge.org/article_S0956796802004331.

[24] D. Walker, "Substructural type systems", in *Advanced Topics in Types and Programming Languages*, 1st ed., The MIT Press, 2004, ch. 1.